Design and Analysis of VLSI Circuits Based on Directed Acyclic Graphs

Kazuyoshi Takagi

January 1999

Abstract

With the recent advances in VLSI technology, the complexity of VLSI circuits has grown far from the level of the manual design, and design automation systems have become indispensable. This thesis discusses graph-based representations of Boolean circuits and functions for design and analysis methods of VLSI logic circuits. We focus on layout synthesis of arithmetic circuits, Boolean function representation for circuit verification and synthesis, and timing analysis of sequential circuits.

In Chapter 2, we deal with layout problems of adder trees used in parallel multiple operand addition. We introduce a class of graphs called p-q dag to represent the connection schemes of adder trees, such as Wallace trees. VLSI layout problem of an adder tree is treated as the minimum cut linear arrangement problem of its corresponding p-q dag. Two algorithms for minimum cut linear arrangement of p-q dags are proposed. One of the two algorithms is based on dynamic programming, and calculates an exact minimum solution within $n^{O(1)}$ time and space, where n is the size of a given graph. The other algorithm is an approximation algorithm which calculates a solution with $O(\log n)$ cutwidth. It requires $O(n \log n)$ time.

In Chapter 3, graph-based representations of Boolean functions are discussed. Ordered Binary Decision Diagrams (OBDDs) are directed acyclic graphs to represent Boolean functions. OBDDs are widely used because of the canonicity and compactness, but alternatives to OBDDs for manipulating still large scale functions are needed for modern VLSI design. We introduce an extension of OBDDs called nondeterministic OBDDs (NOBDDs)

and their restricted forms. NOBDDs are not canonical for representing Boolean functions, but can be more compact than OBDDs. In applications of OBDDs where it is sufficient to check satisfiability, we can use NOBDDs and reduce the required amount of storage. We focus on two particular methods which can be regarded as using restricted forms of NOBDDs and show how the size of OBDDs can be reduced in such forms from the theoretical point of view. First, we consider a method to solve satisfiability problems of combinational circuits where the structure of circuits is used as a key to reduce the NOBDD size. We show that the NOBDD size is related to the cutwidth of circuits. Secondly, we analyze methods that use OBDDs to represent Boolean functions as sets of product terms. We show that the class of functions treated feasibly in this representation strictly contains that in OBDDs and is contained by that in NOBDDs.

In Chapter 4, we focus on exact minimization of Free BDDs (FBDDs) and their application to the design of Pass-transistor Logic (PTL) circuits. FBDD is a well-studied extension of OBDD with free variable ordering on each path, and can be less size than OBDD for the same function. Boolean functions expressed as OBDDs can be directly mapped to PTL circuits, where each node of the OBDDs are replaced by a selector consists of a pair of transistors. The total size of OBDDs (number of nodes) corresponds to the circuit size. We investigate a method using FBDDs instead of OBDDs. We focus on exact minimization of FBDDs and present statistics showing that more than 56% of 616126 NPN-equivalence classes of 5-variable Boolean functions have minimum FBDDs with less size than their OBDDs. We also applied the minimization algorithm of FBDDs to the synthesis of PTL circuits for MCNC benchmarks and found up to 5% size reduction.

In Chapter 5, we describe a method for the timing analysis of sequential circuits based on symbolic state traversal using OBDDs. We focus on the detection of multi-clock paths, whose delay does not affect the decision of the maximum clock frequency. Such paths are typically controlled by waiting states, and the delay time of these paths can be greater than the clock period. We propose a method to detect these paths based on the symbolic state traversal.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

With the recent advances in VLSI technology, the complexity of VLSI circuits has grown far from the level of the manual design, and design automation systems have become indispensable. Design problems of VLSI logic circuits involve combinational problems, and graph-theoretical aspects of the problems are essential for developing efficient design methods.

In this thesis, we deal with graph-based design and analysis methods of VLSI circuits in several design levels. We focus on layout synthesis of arithmetic circuits, Boolean function representation for circuit verification and synthesis, and timing analysis of sequential circuits.

The process of VLSI layout design contains phases of logic cell placement and wire routing among the cells. Automatic placement and routing systems are available nowadays. In the modeling of the layout design problems, grid model is adopted and the layout problems can be treated as graph embedding problems into grids[39]. The major cost functions of embedding are the edge congestion and dilation, which correspond to VLSI area and delay. Many optimization algorithms for graph embedding have been proposed and the algorithms are put into practical use.

For layout generation of arithmetic circuits, the layout algorithms should make use of the regularity of the circuits. When hardware algorithms for arithmetic operations are discussed, the regularity of circuits is often one of the measures for evaluation. The regularity can be formalized as some special properties of the directed acyclic graphs representing the structure of arithmetic circuits. Those properties can be utilized in graph embedding algorithms.

The second topic of this thesis is concerned with graph representations of Boolean functions. For the logic design and analysis of VLSI circuits, representation and manipulation of Boolean functions are major topics. In order to represent Boolean functions compactly, directed acyclic graphs called Ordered Binary Decision Diagrams (OBDDs)[1, 5] have been developed.

OBDDs have a reduced canonical form for each Boolean function. Many Boolean functions we deal with in real design can be represented by OBDDs of feasible size, and efficient operations on OBDDs are possible. Because of these properties, OBDDs are widely used in logic design.

A family of OBDDs can be regarded as a computational model. An OBDD is a restricted form of branching programs. The computational power of size-bounded branching programs have been studied[24], and that of OBDDs have also been studied[19, 33]. It is proved that logarithm of the size of branching programs roughly corresponds to the space complexity of Turing machines. This relationship also holds between OBDDs and online Turing machines.

With the increase of available amount of computer storage, feasible size of OBDDs have increased. However, manipulation of still large scale Boolean functions are required, and new data structures have been sought. Several attempts have been done to overcome the limitation of OBDDs. Some of them propose extensions of OBDDs and others use OBDDs in different ways than usual function representation. Theoretical analysis of such approaches is important for understanding the nature of the data structures.

Another topic on OBDDs is their direct implementation into transistor networks. Recently, Pass-Transistor Logic(PTL) circuits have been paid attention for the potential of low-power and high-speed circuit implementation compared to CMOS circuits. An OBDD can be regarded as a network of 2-1 selector. Selector networks can be implemented in PTL circuits by replacing each selector by a pair of transistors.

Design flow for PTL circuits based on OBDDs have been studied[43, 8]. Because the size of the resulting PTL circuit is affected by the size of the OBDD expression, minimization of OBDDs is important for small circuit area. In order to reduce the required amount of storage, Free BDDs (FBDDs)[16] can be used instead of OBDDs. FBDDs are well-studied extension of OBDDs with free variable ordering on each path, and can be less size than OBDDs. In order to make full use of FBDD expression, synthesis and optimization methods for FBDD-based PTL circuit design are expected.

The last topic of this thesis is concerned with timing analysis of sequential circuits based on symbolic state traversal using OBDDs. Symbolic state traversal of sequential circuits[14, 10, 7] is a major application of OBDDs, where the set of states and the state transition relation are expressed by OBDDs, and state transitions are executed by operations on OBDDs. The timing verification is basically the computation of delay time of each path between registers in the circuits, and the symbolic state traversal method captures the dynamic behavior of each path.

1.2 Objectives and Results of the Thesis

In Chapter 2, we deal with layout problems of adder trees used in parallel multiple operand addition. Multiple operand addition is ubiquitous in various digital systems, and multiplication is also performed by multiple operand addition of partial products.

We introduce a class of graphs called p-q dags. A p-q dag represents the connection schemes of adder trees, such as Wallace trees, and the VLSI layout problem of an adder tree is treated as the minimum cut linear arrangement problem of its corresponding p-q dag. Two algorithms for minimum cut linear arrangement of p-q dags are proposed.

One of the two algorithms is based on dynamic programming. It calculates an exact minimum solution within $n^{O(1)}$ time and space, where n is the size of a given graph. The other algorithm is an approximation algorithm which calculates a solution with $O(\log n)$ cutwidth. It requires $O(n \log n)$ time.

In Chapter 3, graph-based representations of Boolean functions are discussed. OBDDs are directed acyclic graphs to represent Boolean functions. OBDDs are widely used because of the canonicity and compactness, but alternatives to OBDDs for manipulating still large scale functions are needed for modern VLSI design. We introduce an extension of OBDDs called nondeterministic OBDDs (NOBDDs) and their restricted forms. NOBDDs are not canonical for representing Boolean functions, but can be more compact than OBDDs. In applications of OBDDs where it is sufficient to check satisfiability, we can use NOBDDs and reduce the required amount of storage.

It is known that the size of OBDDs is related to the space complexity of deterministic online Turing machines. We show that this relationship also holds between NOBDDs and nondeterministic online Turing machines.

We focus on two particular methods which can be regarded as using restricted forms of NOBDDs and show how the size of OBDDs can be reduced in such forms from the theoretical point of view. First, we consider a method to solve satisfiability problems of combinational circuits where the structure of circuits is used as a key to reduce the NOBDD size. We show that the NOBDD size is related to the cutwidth of circuits. Secondly, we analyze methods that use OBDDs to represent Boolean functions as sets of product terms. We show that the class of functions treated feasibly in this representation strictly contains that in OBDDs and is contained by that in NOBDDs.

In Chapter 4, we focus on exact minimization of Free BDDs (FBDDs) and their application to the design of Pass-transistor Logic (PTL) circuits.

In design flow for PTL circuits based on OBDDs, Boolean functions expressed as OBDDs are directly mapped to PTL circuits, where each node of the OBDDs is replaced by a selector consists of a pair of transistors. The total size of OBDDs (number of nodes)

corresponds to the circuit size.

We investigate a method using FBDDs instead of OBDDs. We focus on exact minimization of FBDDs and present statistics showing that more than 56% of 616126 NPN-equivalence classes of 5-variable Boolean functions have minimum FBDDs with less size than their OBDDs. We also applied the exact minimization algorithm of FBDDs to the synthesis of PTL circuits for MCNC benchmarks and found up to 5% size reduction.

In Chapter 5, we describe a method for the timing analysis of sequential circuits based on symbolic state traversal using OBDDs. The clock frequency of a sequential logic circuit is decided based on the maximum delay of the combinational parts of the circuit. Therefore, the precise estimation of the maximum delay is important in deciding the proper clock frequency. We focus on the detection of multi-clock paths, whose delay does not affect the decision of the maximum clock frequency. Such paths are typically controlled by waiting states, and the delay time of these paths can be greater than the clock period. We propose a method to detect these paths based on the symbolic state traversal.

In Chapter 6, the conclusion of this thesis and future works are stated.

Chapter 2

Minimum Cut Linear Arrangement of p-q Dags

2.1 Introduction

Adder trees, such as Wallace trees, are schemes for parallel multiple operand addition. Multiple operand addition is ubiquitous in various digital systems, and multiplication is also performed by multiple operand addition of partial products. Since multiplication plays an important role, development of a high-speed VLSI multiplier has been one of the major research topics in the area of VLSI systems.

Parallel multiple operand adder schemes, and also parallel multiplier schemes, can be put into roughly two categories: those with an adder tree, and, those based on iterative array. Iterative array schemes have been mainly used in fabrication because they have simpler structure compared to adder trees. However, as the number of operands N increases, adder tree schemes become much faster because they operate in $O(\log N)$ time, while iterative array schemes operate in $O(N)$ time. For high-speed and large-scale operation, therefore, adder tree schemes are attractive. Recently, these schemes came into use for implementation.

Layout generation of theoretically fastest adder trees has been considered to be diffi-

cult because their connection network is irregular and complicated. Adder trees occupy considerable area in multiple operand adders and multipliers, and the placement of parallel counters heavily affects the area. Therefore, it is important to develop an efficient layout method which can treat their complicated connections.

Several efficient layouts for circuits with fast adder tree have been shown. For example, [31] shows a layout for a multiplier with Wallace tree. [34] shows that for a multiplier with an adder tree which uses 9-2 adders as counter elements. However, each of these layouts is only for a particular scheme with a particular operand length. Although some heuristic methods are shown to be effective for circuit generation of multiplier-accumulators with adder trees[28], no general layout algorithm which is applicable to any of the schemes with arbitrary number of operands has been proposed.

Various adder tree schemes with simpler structure and rather regular connection network have been proposed[40, 46]. However, they trade the depth for the regularity and hence slower by a constant factor than optimal adder tree using the same counter elements. In this chapter, we are concerned with fastest adder trees using given counter elements and propose algorithms which are useful for the layout generation of them.

With the increasing demand for development of ASIC's including a high-speed parallel multiplier with certain operand length, development of a general layout method of such multipliers has become more and more important. We are concerned with such a general method.

We regard the VLSI layout problems as the graph embedding problems. Since adder trees have bit slice structure, we may only consider the layout of a bit slice. We can treat the layout problems of a bit slice as graph embedding problems to one-dimensional mesh, i.e., linear arrangement problems of a graph. We consider the minimum cut linear arrangement problem of graphs which corresponds to the area minimization problem.

In order to represent the connection scheme of adder trees, we introduce a class of graphs called p-q dags. A p-q dag is a directed acyclic graph which corresponds to an adder tree with p-in q-out counter elements. Wallace tree can be represented by a 3-2 dag. We reduce the VLSI area minimization problem of adder trees to the minimum cut linear arrangement problem of its corresponding p-q dags, and propose two algorithms for this linear arrangement problem. One of them is based on dynamic programming. For fixed p and q, it calculates an exact minimum solution within time and space both proportional to $n^{\log_{p/q}(2p+q)}$, where n is the size of a given graph which grows linearly in the number of operands. The other algorithm is an approximation algorithm which calculates a solution with $O(\log n)$ cutwidth. It requires $O(n \log n)$ time.

This chapter is organized as follows: Section 2 contains preliminaries and definition of p-q dags to represent the adder tree structures. In section 3, we observe the basic properties of the linear arrangement of p-q dags. Section 4 gives an algorithm based on dynamic programming. Section 5 gives an approximation algorithm. Section 6 is a conclusion.

2.2 Linear Arrangement Problems and p-q Dags

2.2.1 Minimum Cut Linear Arrangement Problems

Let $G = (V, E, \delta)$ be a directed graph, where V is a set of vertices and E is a set of edges. $\delta = (\delta_-, \delta_+)$, where $\delta_-, \delta_+ : E \to V$, is the incidence function. Edge e 'starts' at the vertex $\delta_-(e)$ and 'ends' at the vertex $\delta_+(e)$. Multiple edges can be treated explicitly in this description.

In linear arrangement of a directed graph, we treat the graph as an undirected graph, by regarding each $(\delta_-(e), \delta_+(e))$ as an unordered pair. Let $G = (V, E, \delta)$ be a finite undirected graph. A linear arrangement of G is a bijection $L : V \to \{1, 2, \cdots, |V|\}$.

A partial linear arrangement L' of G is a linear arrangement of a subgraph G' of G (See Figure 2.1.) Let $G' = (V', E', \delta')$. $|V'|$ is called the length of L' and denoted by $length(L')$. In order to simplify the notation, let $L'(v) = \infty$ for $v \in V \setminus V'$. The dangling cut of L' is the set of edges in E between V' and $V \setminus V'$. Let L' and L'' be partial linear arrangements of a graph G. L'' is said to be an extension of L', denoted by $L' \preceq L''$, if

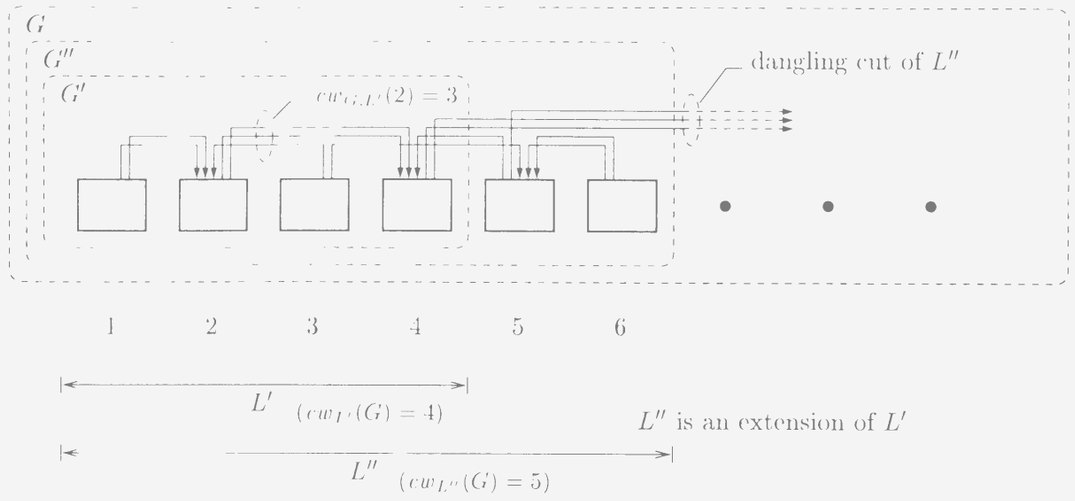


Figure 2.1: Partial linear arrangements of a graph.

$length(L') \leq length(L'')$ and for all $v \in V$ s.t. $L'(v) \neq \infty$, $L'(v) = L''(v)$.

The cutwidth of partial linear arrangement L' of G at position i $(1 \leq i \leq length(L'))$, denoted by $cw_{G,L'}(i)$, is the quantity

$$|\{e \in E \mid \delta(e) = \{u, v\}, \text{ s.t. } L'(u) \leq i < L'(v)\}|.$$

The cutwidth of partial linear arrangement L' of graph G, denoted by $cw_{L'}(G)$, is

$$\max_{1 \leq i \leq length(L')} cw_{G,L'}(i).$$

The cutwidth of graph G, denoted by $cw(G)$, is the minimum cutwidth $cw_L(G)$ of all linear arrangements L of G.

Minimum cut linear arrangement problem of undirected graphs (MINCUT for short) is defined as [15]: "Find a linear arrangement of a given graph with minimum cutwidth."

The following results on MINCUT problem are known.

- MINCUT is NP-Complete for general graphs [15].

- For graphs with cutwidth k, there is an $O(n^k)$ time algorithm for MINCUT [17].
- For graphs with maximum vertex degree 3, MINCUT is still NP-Complete [21].
- For trees, there is an $O(n \log n)$ time algorithm for MINCUT [42].
- For complete k-ary trees, there is a linear time algorithm for MINCUT [20].

2.2.2 p-q Dags and Their Basic Properties

In general, a multiplier with an adder tree consists of three parts, namely, a partial product generator, an adder tree for partial product accumulation, and a carry-propagate-adder for carry assimilation. The first part, the partial product generator, is a circuit to generate a whole partial product matrix in parallel. It is basically a matrix of 1-by-1-bit multiplier, i.e., AND gates. Recoding methods, such as modified Booth recoding, are often used here in order to reduce the size of the resulting matrix. The partial products are summed up to two numbers in the next part, i.e., in the adder tree.

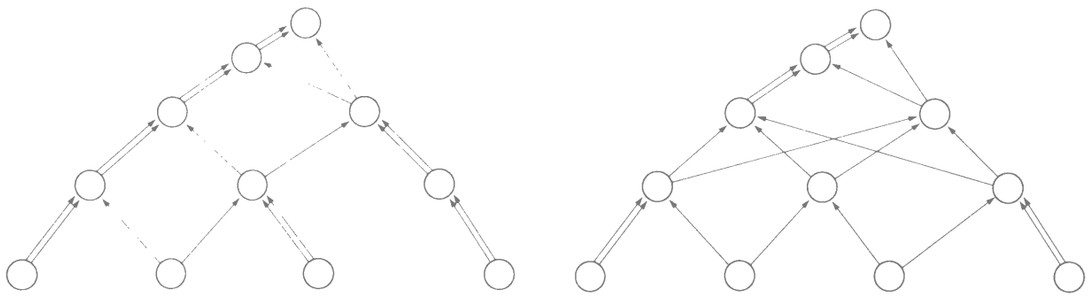
Wallace tree[41] is a well-known adder tree scheme. In Wallace tree, the basic element is a 1-bit full adder which produces a 2-bit binary number from three bits with the same weight. A 1-bit full adder can be regarded as a 3-2 counter. In general, generalized counters[35] can be used as the basic elements to construct an adder tree.

In order to represent the connection scheme of a bit slice of adder trees, we introduce "p-q dags."

Definition 2.1 *A p-q dag is a directed acyclic graph satisfying the following conditions:*

- *There is one vertex, called a root vertex, with indegree at most p and outdegree 0.*
- *The other vertices have indegree at most p and outdegree exactly q.*

Multiple edges are allowed. A vertex with indegree 0 is called a leaf. The height of a p-q dag is the maximum length, i.e., the number of edges, of the directed paths. □



(a) (b)

Figure 2.2: Complete 3-2 dags of height 4.

3-2 dags of height 4 is shown in Figure 2.2. The root vertices are at the top. All edges are drawn upward.

We are interested in the fastest accumulation schemes, i.e., the schemes that accumulate operands in minimum computation time (h), for a given type of circuit elements (p and q) and the number of operands (N). For the sake of simplicity, we prove theorems only for the cases that N is the maximum number for a value of h. We can treat such cases by complete p-q dags defined below. The theorems in this chapter can be extended to the case that N is not a maximum.

Definition 2.2 A complete p-q dag of height h is a graph with the maximum number of vertices among p-q dags of height h. □

A complete p-1 dag is the complete p-ary tree. This definition is a natural extension of the complete tree. Note that for given p, q and h, complete p-q dag of height h is not unique in general. This corresponds to the fact that there can be various order of addition. Both of the two dags in Figure 2.2(a) and (b) are actually complete 3-2 dags of height 4.

There can be various connection pattern for p, q and h, thus various order of addition.

However, because addition is associative and commutative, any complete p-q dag represents proper adder tree. Thus we can select a p-q dag with good nature for embedding, in generating VLSI layout of corresponding adder trees.

Let $G = (V, E, \delta)$ be a complete p-q dag, and $v_0 \in V$ be the root vertex. The depth of a vertex $v \in V$ is the maximum path length from v to v_0 and denoted by $depth(v)$. Let V_d be the set of all vertices of depth d.

Let E_d^- be the set of the edges out of depth d, i.e.,

$$E_d^- = \{e \in E \mid depth(\delta_-(e)) = d\},$$

and E_d^+ be the set of the edges into depth d, i.e.,

$$E_d^+ = \{e \in E \mid depth(\delta_+(e)) = d\}.$$

An edge $e \in E$ is called a leaping edge if

$$depth(\delta_-(e)) - depth(\delta_+(e)) > 1.$$

Let $R_d^-(\subseteq E_d^-)$, and $R_d^+(\subseteq E_d^+)$ be the set of leaping edges out of, and into depth d, respectively. The set of leaping edges through depth d is the set

$$R_d = \bigcup_{i=0}^{d-1} \bigcup_{j=d+1}^{h} R_i^+ \cap R_j^-.$$

Here we show basic properties of p-q dags.

Proposition 2.1 For a complete p-q dag of height h,

(a) for each d, $|R_d^+| \leq q-1$, $|R_d^-| \leq q-1$, and $|R_d| \leq q-1$,

(b) for each d, $\frac{p-q+1}{p}\left(\frac{p}{q}\right)^d \leq |V_d| \leq \left(\frac{p}{q}\right)^d$,

and

(c) $\frac{p-q+1}{p}\frac{(p/q)^{h+1}-1}{p/q-1} \leq |V| \leq \frac{(p/q)^{h+1}-1}{p/q-1}$.

Proof : (a) If $|R_d| \geq q$, a vertex can be inserted at level d without changing the height. This is contrary to the completeness. Thus the last inequality of (a) holds. The others are derived from $|R_{d-1}| < |R_d|$ and $|R_{d+1}| \leq |R_d|$.

(b) $|V_0| = 1$ and $|R_0| = 0$. For fixed p and q, the infinite sequence $(|V_0|, |V_1|, |V_2|, \ldots)$ is unique, that is, the following recurrence equations hold.

$$\begin{aligned} |V_d| &= \lfloor (p\,|V_{d-1}| + |R_{d-1}|)/q \rfloor, \\ |R_d| &= (p\,|V_{d-1}| + |R_{d-1}|) \bmod q, \end{aligned}$$

or, equivalently,

$$q\,|V_d| + |R_d| = p\,|V_{d-1}| + |R_{d-1}|$$

Therefore,

$$\begin{aligned} |V_d| &= \frac{p}{q}\,|V_{d-1}| + \frac{|R_{d-1}| - |R_d|}{q} \\ &= \left(\frac{p}{q}\right)^d |V_0| + \sum_{i=1}^{d} \left(\frac{p}{q}\right)^{d-i} \frac{|R_{i-1}| - |R_i|}{q} \\ &= \left(\frac{p}{q}\right)^d - \sum_{i=1}^{d-1} \left(\frac{p}{q}\right)^{d-i} \frac{p-q}{pq}\,|R_i| - \frac{|R_d|}{q}. \end{aligned}$$

Using property (a),

$$\left(\frac{p}{q}\right)^d - \sum_{i=1}^{d-1} \left(\frac{p}{q}\right)^{d-i} \frac{p-q}{pq}\,(q-1) - \frac{q-1}{q} \leq |V_d| \leq \left(\frac{p}{q}\right)^d,$$

which leads to the inequality of (b).

(c) : This is derived directly from (b) and $V = \bigcup_{d=0}^{h} V_d$. □

2.2.3 p-q Dags and Connection Scheme of Adder Trees

We focus on the layout of the adder tree part. Typically, the adder tree part account for the most of the circuit and their layout heavily affects the chip area. We are concerned with adder trees which are theoretically fastest, including constant factor, among adder trees using given counter elements. As fastest adder trees have complex connections, they

are said to be hard to fabricate on VLSI chips. Our algorithms can be used as a basis for generating the layout of adder trees with any type of generalized counter efficiently.

Adder trees have bit slice structure. We assume that basic circuit components are parallel counters. We can construct a layout of an adder tree from a linear layout of a bit slice. In order to treat the problem formally, we adopt the grid model[39] as a VLSI circuit model. We can obtain a two-dimensional layout by

(1) arranging parallel counters in a bit slice in a column, and,

(2) placing copies of the column iteratively in a row and making connections.

In order to obtain a layout of an adder tree with small area, we have to construct a layout of the bit slice with small width in (1). In (2), we can construct a layout by either (a) placing copies of the same bit slice with enough number of counters iteratively and then reducing redundant counters, or, (b) placing possibly different bit slices with number of counters required for each row. If extra layers for wires are available, the wires can be routed on them above the parallel counters. Otherwise, they are routed between bit slices of parallel counters. In both cases, the bit slice width is proportional to both the size of the counter cells and the number of tracks used for wire routing. The cell size can be considered as a constant, while the number of tracks for wires heavily depends on the layout. Once the placement of parallel counters is fixed, the wire routing can easily be obtained. Therefore, the placement of the counter elements in a bit slice is the key problem of layout of adder trees.

In general, a generalized counter[35] which sums up r k-bit numbers to s d-bit numbers, where $r > s$ and $k < d$, can be used as a basic element of an adder tree. The connection network of a bit slice (of k bits) of an adder tree made up of the generalized counters is represented by an r-$(\lceil \frac{d}{k} \rceil s)$ dag, since a d-bit number is fed to $\lceil \frac{d}{k} \rceil$ counters. When the number of leaves of a p-q dag is l, the corresponding adder tree accumulates lp operands. The height h of the dag corresponds to the computation time of the adder tree.

Once a linear arrangement of a p-q dag is obtained, we can construct a layout of

adder tree. Figure 2.3(b) is an example of linear arrangement of the 3-2 dag shown in Figure 2.3(a). This arrangement achieves cutwidth 5, which is the minimum. A layout of bit slices of 13-input adder tree shown in Figure 2.3(c) is derived from this arrangement. A box in the figure is a full adder which reduces 3 bits of the same weight to a 2-bit number. A broken line box shows a bit slice. Two wires from outputs of a full adder are drawn according to the edges which start from the corresponding vertex. One output (sum) is fed to a full adder in the same bit slice and the other (carry) is fed to one in the next bit slice. In this case, it is sufficient to feed each output to the next bit slice. When $d > 2k$, some of the outputs should be fed to bit slices which are not adjacent using feed through wires. We can put enough number of copies of this bit slice and reduce redundant counters to construct a layout of a 13-input multiple operand adder.

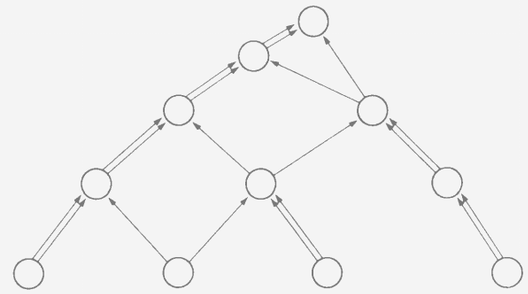
2.3 Minimum Cut Linear Arrangement of p-q Dags

In this section, we define problem PQMINCUT based on the problem MINCUT of complete p-q dags and show properties of this problem.

For defining PQMINCUT, we ignore the directions of edges of p-q dags when we mention cutwidth. Furthermore, we treat all complete p-q dags collectively in order to find a dag with good nature for MINCUT. That is, for given p, q and h, minimum cutwidth is taken over arrangements not only of a fixed complete p-q dag, but all complete p-q dags of height h. Formally, PQMINCUT is defined as follows.

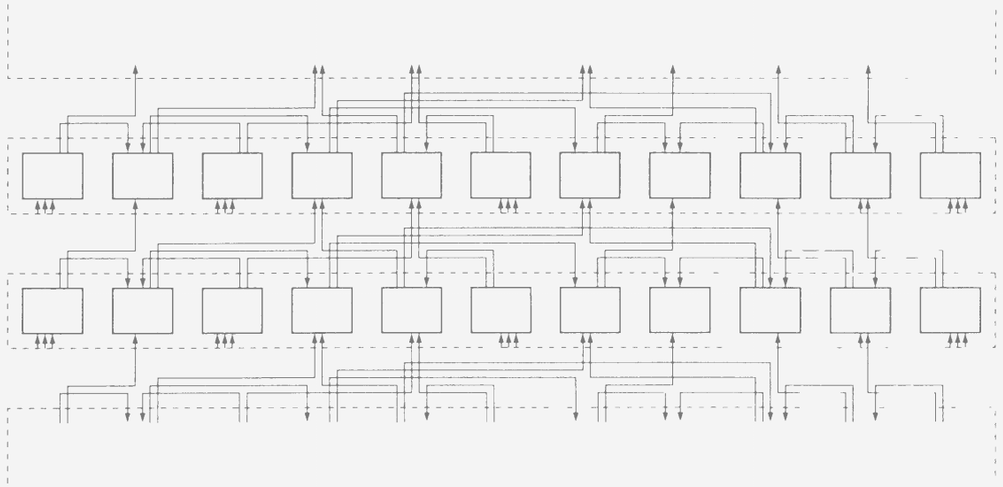
Definition 2.3 Problem PQMINCUT : "For given p, q and h, find a complete p-q dag G of height h and its linear arrangement L, which provide smallest cutwidth $cw_L(G)$." □

In general, the search space of PQMINCUT for given p, q and h is much larger than that of MINCUT for a complete p-q dag of height h. However, the problem PQMINCUT is easier to treat as compared with MINCUT for a p-q dag, because of the regular structure of PQMINCUT. We look into the properties of PQMINCUT and make use of them for linear arrangement algorithms.



(a)

(b)



(c)

Figure 2.3: A layout of 13-input adder tree.

We now see the properties of the minimum cut linear arrangement of complete p-q dags.

There are $n!$ linear arrangements for a graph with n vertices. Therefore, the search space of the problem PQMINCUT is $n!$ times the number of complete p-q dags of height h. We can use some properties of complete p-q dags to reduce the search space.

The next theorem states that we can reduce the search space of PQMINCUT to graphs G and their arrangements L with the following properties.

(i) G is almost planar, namely, planar except for leaping edges.

(ii) L arranges the vertices in each V_d in the 'planar order,' i.e., the fixed order for each depth of the graph.

In other words, for any arrangement L of a p-q dag of height h, there is an arrangement of a p-q dag of height h (not necessary the original dag), which has the above properties, with cutwidth not larger than that of L.

Theorem 2.1 For given p, q, and h, there is a solution of PQMINCUT, i.e., a graph $G = (V, E, \delta)$ and its arrangement L, with the following property.

For each pair of edges $e_1, e_2 \in E$, if

(a) $depth(\delta_+(e_1)) < depth(\delta_-(e_2))$ and
$depth(\delta_+(e_2)) < depth(\delta_-(e_1))$.

then,

(b) either

(b1) $L(\delta_-(e_1)) \leq L(\delta_-(e_2))$ and
$L(\delta_+(e_1)) \leq L(\delta_+(e_2))$, or,

(b2) $L(\delta_-(e_2)) < L(\delta_-(e_1))$ and
$L(\delta_+(e_2)) \leq L(\delta_+(e_1))$.

Proof : Let $G_0 = (V, E, \delta_0)$ be a complete p-q dag of height h. Let L be any linear arrangement with cutwidth w of G_0. We can construct a complete p-q dag $G = (V, E, \delta)$, with cutwidth at most w under linear arrangement L, having the above property.

First, let $G := G_0$. We take all pairs of edges of G satisfying (a), and redraw the edges if needed. The procedure is shown in Figure 2.4. V, E, and L are not changed in this procedure.

```
procedure Redraw_Edges                                              1
begin                                                               2
   for each pair of edges (e1, e2) satisfying (a) do                3
      if (e1, e2) satisfies neither (b1) nor (b2) then              4
         swap δ+(e1) and δ+(e2)                                     5
end                                                                 6
```

Figure 2.4: Procedure Redraw_Edges.

Let $G_1 = (V, E, \delta_1)$ be the graph G at a point of time in this procedure. It is transformed to graph $G_2 = (V, E, \delta_2)$ after one execution of line 5.

Suppose that (e_1, e_2) in G_1 satisfies neither (b1) nor (b2). The operation in line 5 makes (e_1, e_2) in G_2 satisfy (b1) or (b2).

This operation does not change indegree and outdegree of the vertices. Because of the condition (a), G_2 remains acyclic and the height does not change. Thus the graph G_2 is a complete p-q dag of height h.

Suppose that (b1) is satisfied after the operation. Figure 2.5 shows all the four cases of the transformation except for symmetry. The horizontal lines represent an arrangement and only the four vertices and two edges under consideration are drawn.

Because the cutwidth of L of G_2 differs from that of G_1 only in the location of edges

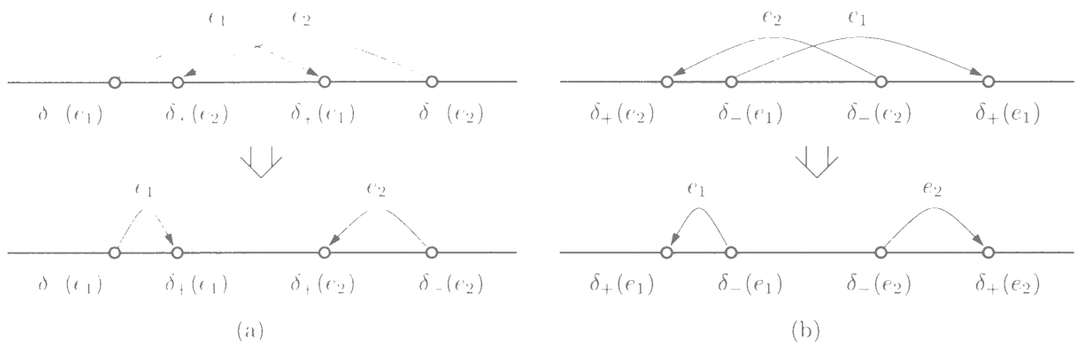


Figure 2.5: Operations in procedure Redraw_Edges.

e_1 and e_2.

$$cw_{G_2,L}(i) = \begin{cases} cw_{G_1,L}(i) - 2, \\ \quad \text{if } \max\{L(\delta_-(e_1)), L(\delta_{2+}(e_1))\} \le i < \min\{L(\delta_-(e_2)), L(\delta_{2+}(e_2))\}, \\ cw_{G_1,L}(i), \text{ otherwise.} \end{cases}$$

Let 'total squared edge length' be the quantity

$$Q(G, L) = \sum_{e \in E} (L(\delta_-(e)) - L(\delta_+(e)))^2.$$

Q decreases by this operation, i.e., $Q(G_2, L) < Q(G_1, L)$.

Therefore, the operation does not increase the cutwidth and does decrease the total squared edge length Q. The case that (b2) is satisfied is similar.

This operation is applied to all pairs of edges satisfying (a) to obtain final G. Because the total squared edge length decreases by one operation, a graph does not appear more than once in the procedure. For the number of possible graphs is finite, the condition (b) eventually comes to hold for all edge pairs satisfying (a).

Thus we obtain the transformed graph G. Arrangement L of G satisfies the condition, and the cutwidth is at most w. □

Let $L' : V' \to \{1, 2, \ldots, |V'|\}$ $(V' \subseteq V)$ be a partial linear arrangement of a complete p-q dag $G = (V, E, \delta)$ of height h. The out-of-depth-d-cut of L' of G at i $(1 < d \le h, 1 \le i \le |V'|)$ is the set

$$\begin{aligned} cut^+_{G,L'}(d, i) \;=\; & \{e \in E \mid depth(\delta_-(e)) = d \\ & \text{and } L'(\delta_-(e)) \le i < L'(\delta_+(e))\} \end{aligned}$$

The in-to-depth-d-cut of L' of G at i $(0 \le d \le h - 1, 1 \le i \le |V'|)$ is the set

$$\begin{aligned} cut^-_{G,L'}(d, i) \;=\; & \{e \in E \mid depth(\delta_+(e)) = d \\ & \text{and } L'(\delta_+(e)) \le i < L'(\delta_-(e))\} \end{aligned}$$

Recall that for all $v \in V \setminus V'$, $L'(v) = \infty$. We can describe the cutwidth of a linear arrangement L at i ($1 \leq i \leq |V|$) using these sets as follows.

$$cw_{G,L}(i) = \sum_{d=1}^{h} (|cut^{+}_{G,L}(d,i)| + |cut^{-}_{G,L}(d-1,i)|)$$

Note that when (G, L) has the property of Theorem 2.1, at least one of the sets $cut^{+}_{G,L}(d,i)$ and $cut_{G,L}(d-1,i)$ is empty. This is because, if there are edges $e_1 \in cut^{+}_{G,L}(d,i)$ and $e_2 \in cut_{G,L}(d-1,i)$, they satisfy (a) in Theorem 2.1, but can satisfy neither (b1) nor (b2).

Theorem 2.1 points out that we only have to treat the case that each vertex set V_d is arranged in the 'planar order.' We can further assume that the 'slippage' between each pair of depth is small.

Theorem 2.2 For given p, q, and h, there is a solution of PQMINCUT, i.e., a graph $G = (V, E, \delta)$ and its linear arrangement L, such that, for any $i \in \{1, 2, \ldots, |V| - 1\}$ and $d \in \{0, 1, \ldots, h-1\}$,

$$\begin{aligned} |cut^{-}_{G,L}(d,i)| &\leq p, \text{ and} \\ |cut^{+}_{G,L}(d+1,i)| &\leq p + |R^{-}_{d+1}|. \end{aligned}$$

Proof : Let $G_0 = (V, E, \delta)$ be a complete p-q dag of height h. Let L_0 be a linear arrangement with cutwidth w of G_0. We can assume that L_0 and G_0 have the property of Theorem 2.1. We can construct an arrangement L of G, with cutwidth at most w, satisfying the above inequalities.

Let L^R be the reverse arrangement of L defined as:

$$L^R(v) = |V| + 1 - L(v) \text{ for all } v \in V.$$

We rearrange vertices, from the depth $h-1$ down to the depth 0, so that each vertex is placed near its adjacent vertices. Figure 2.6 is an algorithm for rearrangement.

Let us look into operations on v of depth d. Suppose that (G, L) has the property of Theorem 2.1 and let $C_1 = cut_{G,L}(d, L(v))$ and $C_2 = cut_{G,L^R}(d, L^R(v))$.


```
procedure Rearrange_Vertices                                          1
begin                                                                 2
  G := G_0;  L := L_0                                                 3
  for d from h − 1 downto 0 do                                        4
    for each vertex v ∈ V_d do                                        5
      begin                                                           6
        while |cut⁻_{G,L}(d, L(v))| > p do                            7
          begin /* move v to the right */                             8
            find u such that L(u) = L(v) + 1                          9
            L(v) := L(v) + 1;  L(u) := L(u) − 1                       10
            reconstruct (G, L) by procedure Redraw_Edges              11
          end                                                         12
        while |cut⁻_{G,L^R}(d, L^R(v))| > p do                        13
          begin /* move v to the left */                              14
            find u such that L(u) = L(v) − 1                          15
            L(v) := L(v) − 1;  L(u) := L(u) + 1                       16
            reconstruct (G, L) by procedure Redraw_Edges              17
          end                                                         18
      end                                                             19
end                                                                   20
```

Figure 2.6: Procedure Rearrange Vertices.

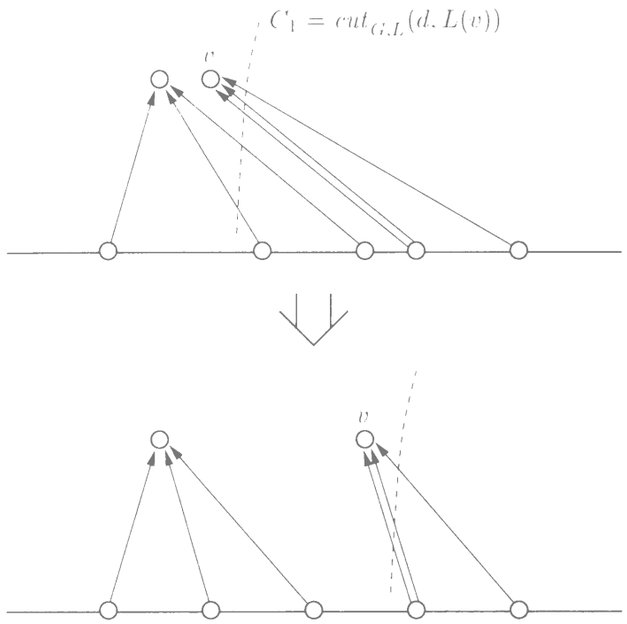


Figure 2.7: Operations in procedure Rearrange_Vertices.

When the condition of the first 'while' loop holds, i.e., $|C_1| > p$, then $C_2 = \emptyset$. This can be proved as follows: If there exists an edge $e_2 \in C_2$, then e_2 and any $e_1 \in C_1$ satisfy the condition (a) of Theorem 2.1, $L(\delta_-(e_2)) < L(\delta_-(e_1))$, and $L(\delta_+(e_1)) \leq L(v) \leq L(\delta_+(e_2))$. The former two imply $L(\delta_+(e_2)) \leq L(\delta_+(e_1))$ from Theorem 2.1 (b2), which with the last leads to $\forall e \in C_1 \cup C_2, \delta_+(e) - v$, and therefore $|C_1| \leq p$.

Figure 2.7 shows a situation when $|C_1| > p$. The horizontal lines represent an arrangement and the two vertices above the line are in depth d. Only the edges into depth d are drawn. Because the number of edges from u (found in line 9 of the procedure) is at most q, $|C_2|$ can increase at most q in one pass of the loop. Therefore, $|C_2| \leq q(< p)$ holds when the process exits the first 'while' loop. Hence the second 'while' loop is not executed.

Similarly, if the condition of the second 'while' loop holds, $|C_1| \leq q(< p)$ holds when

the process exits the loop.

Each pass of the loop does not increase the cutwidth of G. The operations for a vertex in depth d do not destroy the inequalities of the theorem for depth $d, d+1, \ldots, h-1$. Therefore, the operations can be applied from depth $h-1$ down to depth 0. Thus we can translate (G, L) with cutwidth at most w, satisfying the first inequality of the theorem.

The edges in $cut^+_{G,L}(d+1, i) \setminus cut^-_{G,L^R}(d, |V| - i)$ are leaping edges from depth $d+1$. Therefore, the next inequality holds.

$$|cut^+_{G,L}(d+1, i)| \leq |cut^-_{G,L^R}(d, |V| - i)| + |R_{d+1}|$$

This leads to the second inequality of the theorem. □

2.4 Algorithms Based on Dynamic Programming

In this section, we describe an algorithm based on dynamic programming for PQMINCUT. The algorithm is fed p, q and h and yields a linear arrangement of a graph with the minimum cutwidth.

Dynamic programming is a bottom-up method to solve various optimization problems with decomposable structure. The MINCUT problem has a structure such that partial linear arrangements with small cutwidth lead to the solution of the MINCUT problem. In order to utilize this structure, we define an equivalence relation among p-q dags and their linear arrangements. Let L_a and L_b be two partial linear arrangements of p-q dags of height h, G_a and G_b, respectively, where $length(L_a) > 0$ and $length(L_b) > 0$. (G_a, L_a) and (G_b, L_b) are called 'dangling cut equivalent,' if the next equations hold:

for all $1 \leq d \leq h$,
$|cut^+_{G_a,L_a}(d, length(L_a))| = |cut^+_{G_b,L_b}(d, length(L_b))|$,
and
$|cut^-_{G_a,L_a}(d-1, length(L_a))| = |cut^-_{G_b,L_b}(d-1, length(L_b))|$.

Note that this implies $length(L_a) = length(L_b)$. G_a and G_b are not necessarily isomorphic. Let $[(G, L)]$ be the dangling cut equivalence class to which (G, L) belongs. Let $[(G, L)]_{min}$ be the set of pairs in $[(G, L)]$ which gives the minimum cutwidth.

The algorithm shown below searches a linear arrangement L_{min} of a p-q dag G_{min} which provides the minimum cutwidth. The search space of the algorithm is the set of dangling cut equivalence classes $\{[(G, L)]\}$ with (G, L) having the properties in Theorems 2.1 and 2.2.

$[(G, L)]$ can be represented by a set of $2h$ values of cutwidths out-of and in-to depths 0 to h, i.e., an h-tuple

$$(cio_1, cio_2, \ldots, cio_h)$$

where, each cio_d $(1 \leq d \leq h)$ is a pair

$$(|cut^{+}_{G,L}(d, length(L))|, |cut^{-}_{G,L}(d-1, length(L))|).$$

As we have noted, at least one of the values of each cio_d is 0 for (G, L) having the property in Theorems 2.1. Thus possible pair of values of cio_d for each d is at most $(p + (p + q - 1) + 1) = (2p + q)$ from Theorem 2.2.

In order to find the minimum cutwidth, all we have to keep track of is

$$cw_{[(G,L)]} = \min_{(G',L') \in [(G,L)]} cw_{L'}(G')$$

for each $[(G, L)]$. The algorithm starts at the arrangement of length 0. The vertices are arranged from left to right. That is, in the i-th stage, it constructs all $[(G, L_{i+1})]$ $(length(L_{i+1}) = i+1)$ from each $[(G, L_i)]$ $(length(L_i) = i)$ by extending L_i. This procedure is a traversal of the space shown in Figure 2.8 in a breadth-first manner. In this figure, each state is represented by a set of arranged vertices, and the next states with one new vertex each are examined. Figure 2.9 shows a rough sketch of one step. C and C' separating the dag in the figure are dangling cuts of (G, L_i) and (G, L_{i+1}) respectively. Tuples of cutwidths corresponding to C and C' represent $[(G, L_i)]$ and $[(G, L_{i+1})]$ respectively. Here, for each cut C, cutwidths in the left and right of C can be minimized independently.

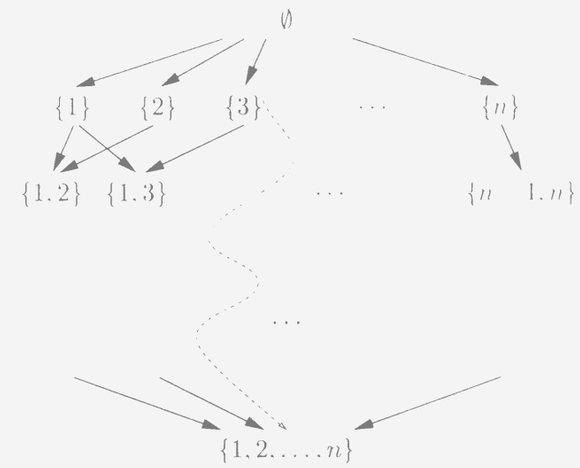


Figure 2.8: Search space of procedure Mincut.

Because of this good property, it is sufficient to keep all possible (G, L_i) of length i with minimum cutwidth, in order to generate (G, L_{i+1}) of length $i+1$ with minimum cutwidth. $cw_{[(G,L_{i+1})]}$ are easily calculated from $cw_{[(G,L_i)]}$. After the $(|V|-1)$-th stage, the minimum cutwidth is obtained as $cw_{[(G,L_{|V|})]}$ $(length(L_{|V|}) = |V|)$.

The procedure is shown in Figure 2.10. At lines 6 and 7, the number of classes $[(G, L_i)]$ $(length(L_i) = i)$ is the number of patterns of $(cio_1, cio_2, \ldots, cio_h)$, which is at most $(2p + q)^h$. At line 8, vertices in each depth are tested, along with all possible combinations of the numbers of leaping edges into and out of them. The number of classes $[(G, L_{i+1})]$ for each $[(G, L_i)]$ for each depth is at most $(2p + q)$, which amounts to $(2p + q)h$ in total for all depths. Thus the number of iterations of innermost loop is $O((2p+q)h(2p+q)^h)$. We can use the quantity $\{|R_d|\}$ and $\{|R^*_d|\}$ in Proposition 2.1 for further reduction of the search space.

We have to keep $cw_{[(G,L_i)]}$ for all $[(G, L)]$. We also have to keep a pair $(G, L) \in [(G, L)]_{min}$ for each $[(G, L)]$ to calculate (G_{min}, L_{min}). $O((2p + q)h(2p + q)^h)$ space is sufficient to store these information.

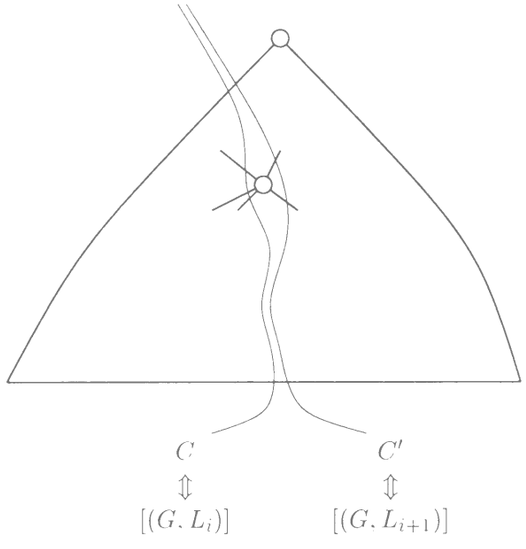


Figure 2.9: One step of procedure Mincut.

Therefore, for fixed p and q, this algorithm spends at most both time and space $O((2p+q)^h h)$, simultaneously.

Together with Proposition 2.1(c), we have the following theorem.

Theorem 2.3 For fixed p and q, MINCUT for complete p-q dags of height h can be solved in $O(n^{\log_{p/q}(2p+q)})$ time and space, where n is the number of vertices. □

An example is shown in Figures 2.11 and 2.12. Figure 2.11 is a complete 3-2 dag of height 10. The edges are directed upward and a bold line represents duplicate edges. Figure 2.12 shows a minimum cut linear arrangement of the graph. The numbers in the vertices are the level, and the ranking in each level counted from left in figure 2.11. We can derive a layout of a multiple operand adder based on Wallace tree scheme which accumulates 141 operands.

If an upper bound of the cutwidth is given, we can further reduce the search space. In the algorithm, if $cw_{[(G,L]}$ is greater than the upper bound, extensions of L need not be ex-


```
procedure Mincut(p, q, h)                                              1
begin                                                                  2
    calculate |V|                                                      3
    cw_[(G,L_0)] := 0 where L_0 is the arrangement of length 0          4
    cw_[(G,L)] := ∞ where L ≠ L_0                                      5
    for i from 0 to |V| − 1 do                                         6
        for each [(G, L_i)] (length(L_i) = i) do                       7
            for each [(G, L_{i+1})] (length(L_{i+1}) = i + 1)
                where ∃G', L', L'', s.t.
                    (G', L') ∈ [(G, L_i)]_min ,
                    (G', L'') ∈ [(G, L_{i+1})], L' ⪯ L'' do             8
            if cw_[(G,L_{i+1})] > cw_{L''}(G') then
                cw_[(G,L_{i+1})] := cw_{L''}(G')                        9
            /* cw_{L''}(G') = max{cw_[(G,L_i)], cw_(G',L'')(i + 1)} */  10
    return (G_min, L_min) ∈ [(G, L_|V|)]_min
        where L_|V| is an arrangement of length |V|                    11
end                                                                    12
```

Figure 2.10: Procedure Mincut.

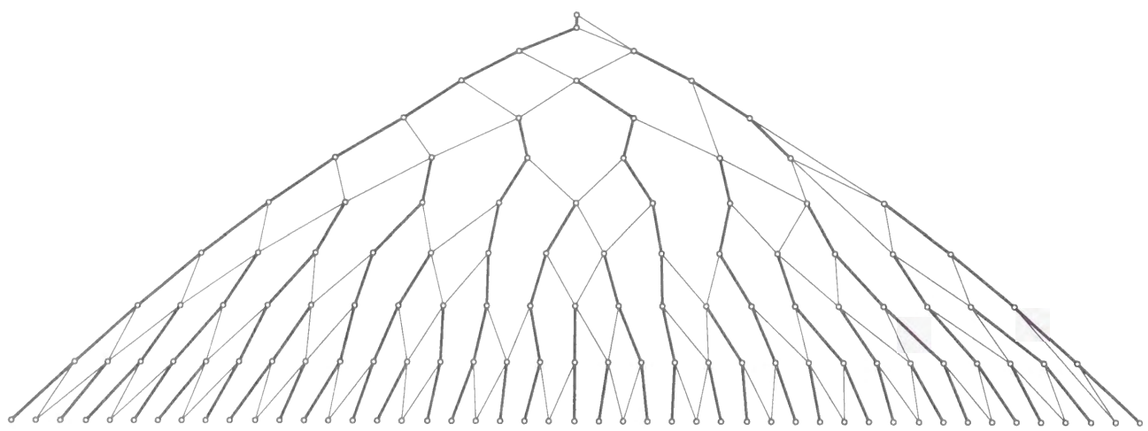


Figure 2.11: A complete 3-2 dag of height 10.

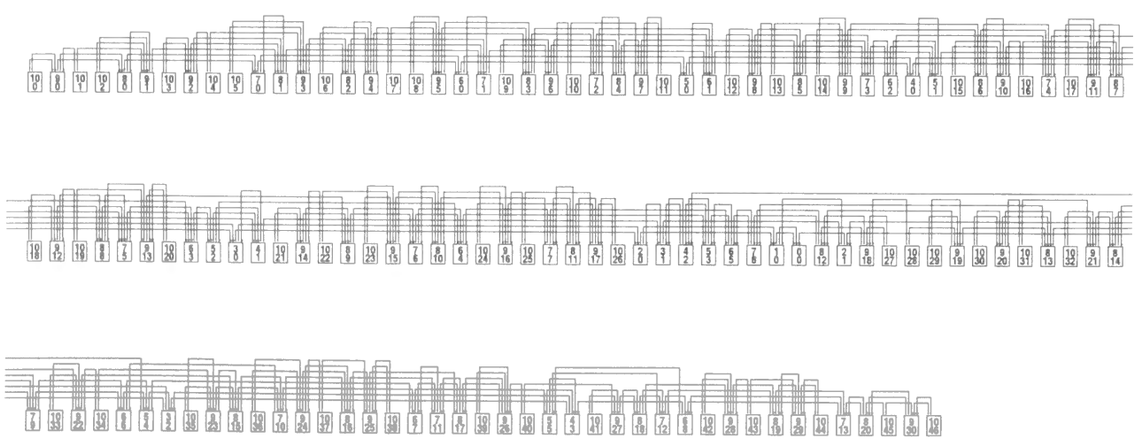


Figure 2.12: A linear arrangement of a complete 3-2 dag of height 10.

amined. From Theorem 2.2, an upper bound $(p+q-1)h$ can be obtained. Approximation Algorithms such as one in the next section are also useful to obtain upper bounds.

This algorithm can be extended for the cases that p-q dags are not complete, though the search space can be larger in such cases.

2.5 Fast Approximation Algorithms

We show another algorithm for MINCUT of p-q dags in this section. This algorithm requires a smaller quantity of time and space than the algorithm in the previous section.

In the algorithm shown in Figure 2.13, the vertices are arranged from left to right. First, a graph $G = (V, E, \delta)$ with property of Theorem 2.1 is generated. Each V_d is arranged in the 'planar order.' In the i-th iteration, vertices in each V_d are examined from $d = h$ down to $d = 0$. If more than the half of the edges which are incident to the vertex v are dangling, v is arranged in the i-th position. If there is no such vertex, the next leaf is arranged. The algorithm yields an arrangement with the property of Theorem 2.2. Therefore, the cutwidth is within a constant factor of the height, i.e., $O(\log n)$.

The generation of G at line 3 is easy. The outer 'for loop' spends $O(h)$ time for an iteration. One iteration arranges exactly one vertex, while the algorithm in the previous section tries several vertices at each depth. Therefore, the algorithm terminates after $n = |V|$ iterations.

Theorem 2.4 For fixed p and q, algorithm Approx_mincut requires $O(n \log n)$ time, where n is the number of vertices. It yields a linear arrangement with cutwidth $O(\log n)$. □

2.6 Conclusion

We have proposed two algorithms for minimum cut linear arrangement of p-q dags. The algorithms give systematic methods to construct VLSI layout of adder trees of any size

```
procedure Approx_mincut(p, q, h)                                          1
begin                                                                     2
   fix a graph G = (V, E, δ) satisfying Theorem 2.1                       3
   calculate V_0, V_1, ..., V_h                                           4
   L := L_0 (the arrangement of length 0)                                 5
   for i from 1 to |V| do                                                 6
      begin                                                               7
         if |cut^+_L(h, length(L))| > ⌊q/2⌋ then d := h                   8
         else                                                             9
            begin                                                         10
               for d from h − 1 to 1 do                                   11
                  if |cut^+_{G,L}(d, length(L))| > ⌊(p+q)/2⌋ then goto add_vertex   12
               if |cut_{G,L}(0, length(L))| > ⌊p/2⌋ then d := 0            13
               else d := h                                                14
            end                                                           15
         add_vertex:  /* extend L; length(L) := i */                     16
         L(v) := i, where v is the leftmost vertex in V_d
               which has not arranged yet                                 17
      end                                                                 18
   return L                                                               19
end                                                                       20
```

Figure 2.13: Procedure Approx_Mincut.

using any counter component. We reduced the layout area minimization problem of adder trees to a graph problem, the minimum cut linear arrangement problem of p-q dags.

The first algorithm finds out an exact minimum solution through dynamic programming approach. We proved that the minimum cut of a complete p-q dag is attained by an arrangement of almost planar graphs with small 'slippage' between each depths. We can reduce the search space based on these properties. For fixed p and q, the algorithm calculates a solution within time and space $O(n^{\log_{p,q}(2p+q)})$ where n is the size of a given graph.

The second algorithm is an approximation algorithm which calculates an arrangement with $O(\log n)$ cutwidth. This algorithm runs in $O(n \log n)$ time.

The layout of adder trees such as Wallace tree have been thought to be difficult because of their complex connection scheme. However, we can construct VLSI layout of those circuits efficiently by means of our algorithms.

We can construct algorithms similar to those shown in this chapter for the problem MINSUM, which minimizes the total edge length, based on dynamic programming. It is easy to see that Theorem 2.1 and Theorem 2.2 for cutwidth also holds for total edge length. The algorithm for MINSUM is obtained by keeping the sums of the edge length of partial linear arrangements, instead of keeping the cutwidths.

Chapter 3

Computational Power of Nondeterministic Ordered Binary Decision Diagrams

3.1 Introduction

Efficient representation and manipulation of Boolean functions are indispensable in various fields of computer science. Ordered Binary Decision Diagrams (OBDDs)[1, 5] are directed acyclic graphs representing Boolean functions, which have been used for various applications because of their good properties: 1) given a total order of the input variables, OBDDs have a reduced canonical form for each Boolean function, 2) many practical Boolean functions can be represented by OBDDs of feasible size, and 3) there are efficient algorithms for Boolean operations on OBDDs.

With the increase of available amount of computer storage, feasible size of OBDDs have been increased. However, requirements for manipulating still larger scale Boolean functions are growing.

In some applications, it is not necessary to represent functions in canonical forms. In this chapter, we introduce Nondeterministic OBDDs (NOBDDs) and observe that we can

handle larger class of functions using NOBDDs, at the cost of canonicity, and that some applications can be viewed as utilizing the power of nondeterminism from theoretical point of view.

A family of OBDDs can be regarded as a computational model. The computational power of size-bounded OBDDs have been studied[19, 33]. They showed that logarithm of the size of OBDDs roughly corresponds to the space complexity of deterministic on-line Turing machines. We show that this relationship also holds between NOBDDs and nondeterministic online Turing machines. Namely, the class of functions computable by polynomial size NOBDDs is identical to the class of functions computable by logarithmic space-bounded nondeterministic online Turing machines.

However, this relationship only gives the limit of applications of NOBDDs in general. In this chapter, we treat two particular methods in practical applications that make use of the expressions using the nondeterminism of OBDDs. We investigate to what extent they utilize the nondeterminism introduced into OBDDs.

First, we treat a method to solve satisfiability problem of combinational circuits using NOBDDs.

The size of OBDDs heavily depends on the total order defined on the input variables. It is known that the OBDD size is closely related to the circuit width, and when a linear arrangement of the circuit with width w is known, a variable ordering which gives OBDD of size $2^w n$ can be derived[4, 23]. The size of NOBDDs depends on the way of introducing nondeterminism, in addition to the variable ordering. In constructing an OBDD in order to decide satisfiability, Hamaguchi et al. proposed a method to decompose the given circuit into subcircuits and decides the variable ordering, based on the circuit structure [18]. This decomposition can be regarded as introducing nondeterminism into the OBDD. In this method, the OBDD size is related to the circuit cutwidth, defined as the cutwidth of the circuit as an undirected graph, which is a two-way counterpart of the circuit width. In this chapter, we investigate which kind of functions can be treated feasibly through this approach. We define the class of functions computable by circuits with logarithmic

cutwidth, and show that this class is strictly contained by the class of functions represented by polynomial size NOBDDs.

Secondly, we treat a method that uses OBDDs to represent Boolean functions as sets of terms[11], i.e., in sum-of-product form. Recently, varieties of OBDDs more suitable for expression and manipulation of sum-of-product form formulas are proposed[25, 45]. In these representations, each path from the root node to the '1' node corresponds to a term. Although OBDDs for these representations are not canonical with respect to the functions, they have one-to-one correspondence with sum-of-product forms. In this chapter, we show that these representations can also be regarded as restricted forms of NOBDDs.

This chapter is organized as follows. Section 2 gives fundamental definitions. In Section 3, we show some methods that can be viewed as utilizing nondeterminism, and observe the computational power of NOBDDs. In Section 4, we investigate methods to construct NOBDDs based on circuit structure and relate the NOBDD size and the circuit cutwidth. In Section 5, we investigate expressive power of OBDDs which represent sum-of-product form. Section 6 is a conclusion.

3.2 Preliminaries

3.2.1 Ordered Binary Decision Diagrams

An OBDD represents a Boolean function over $\{0,1\}^n$. An OBDD is a labeled, directed acyclic graph with a unique source node, called the root node. There are two sink nodes, called the constant nodes, labeled by a Boolean value 0 and 1 respectively. Other nodes are called variable nodes. Each variable node is labeled by one of the n variables and has exactly two outgoing edges, 0-edge and 1-edge respectively. On any path from the root node to a sink node, each variable appears as a label of a node at most once. The order in which the variables appear is consistent among all paths. That is, the set of variables $\{x_1, x_2, \ldots, x_n\}$ is ordered by a bijection $\pi : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$, and if

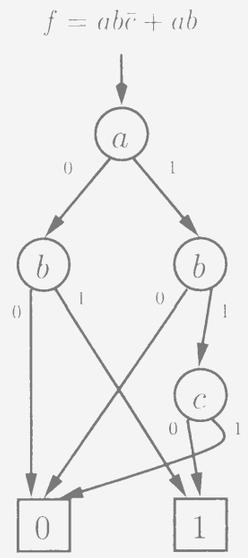


Figure 3.1: An OBDD representing $f = ab\bar{c} + \bar{a}b$.

$\pi(i) \prec \pi(j)$, x_j cannot appear before x_i on any path. An OBDD representing $f = ab\bar{c} + \bar{a}b$ is shown in Figure 3.1.

When a set of value assignment for the input variables is given, the OBDD is traversed from the root node to a constant node according to the assignment. The label of the reached constant node is interpreted as the function value.

When a variable ordering is fixed, OBDDs have reduced canonical form. The reduction is done by merging equivalent nodes and removing redundant nodes. The size, defined as the number of nodes, of reduced OBDDs depends on the variable ordering.

We regard a family of OBDDs $\{D_i\}_{(i\ =\ 0,1,\ldots)}$, where D_i represents a Boolean function over $\{0,1\}^i$, as a computation model which computes a Boolean function over $\{0,1\}^*$.

3.2.2 Complexity Classes of Functions

In this chapter, we treat only languages over $\{0,1\}^*$ and identify a language with its characteristic function.

We cannot directly compare classes of functions based on OBDDs with those based on Turing machines. This is because: i) OBDD model is nonuniform, i.e., we have to provide different OBDD for each input size n, while one Turing machine can handle inputs of all sizes, and ii) OBDD size is affected by the input variable ordering, while the order of input variables is fixed for Turing machine model. In this chapter, we choose nonuniform models and allow variable ordering to be selected arbitrary by adjusting the Turing machine model to the OBDD model. We can discuss similarly in other settings.

We give a definition of the classes of functions $\mathcal{C}/poly$, which are nonuniform counterparts of classes $\mathcal{C}$ based on Turing machines with advice, where $\mathcal{C}$ is L,NL etc. A *Turing machine with advice* is a Turing machine that has access to a special oracle, called *advice*. A Turing machine with advice works as follows. Let $f : \mathbb{N} \to \{0,1\}^*$ be a function. f is called an *advice function*. On input x of size $|x|$, $f(|x|)$ is written on the special read-only *advice tape*, before the computation starts. Then the computation proceeds as usual.

L/poly (*NL/poly*, respectively) is the class of functions computable in logarithmic space bounded deterministic (nondeterministic) Turing machines with advice such that the length of the string $|f(n)|$ is bounded by $n^{O(1)}$. *1-L/poly* and *1-NL/poly* are defined in the same way except that the Turing machines are online, i.e., they can read each bits on the input tape only once in order. Note that the length of the advice tape is not counted in the computation space and the online condition is not applied to the advice tape.

The class of complements of a class $\mathcal{C}$, denoted by *co-*$\mathcal{C}$, is defined as the class $\{A \mid \{0,1\}^* \setminus A \in \mathcal{C}\}$.

Next, we define permutation closure Π, in order to deal with input variable ordering. Let $\mathcal{C}$ be a class of functions defined by online Turing machines. The permutation closure

$\Pi(\mathcal{C})$ is defined as:

$$\{B \mid \exists A \in \mathcal{C}.$$
$$\exists \text{ a family of bijections } \{\pi_n : \{1,\ldots,n\} \to \{1,\ldots,n\}\}_{n=0,1,\ldots}.$$
$$s.t.,$$
$$\forall x_1 x_2 \cdots x_n \in \{0,1\}^n,$$
$$x_1 x_2 \cdots x_n \in B \Leftrightarrow x_{\pi_n(1)} x_{\pi_n(2)} \cdots x_{\pi_n(n)} \in A\}.$$

In other words, while $\mathcal{C}$ is defined by Turing machines which read the input once in order, $\Pi(\mathcal{C})$ can be defined by Turing machines which read the input once in the order which is fixed for each input size. We refer to this kind of machines as $\Pi(\mathcal{C})$ machines.

We are interested in the characterization of functions representable by feasible size OBDDs. Here, our criterion for feasibility is *polynomial size*, i.e., the size within $n^{O(1)}$ where n is the input size. Let *PolyOBDD* be the class of functions representable by OBDDs of polynomial size. The next theorem is known.

Theorem 3.1 [19, 33] $PolyOBDD = \Pi(1\text{-}L/poly)$ □

3.3 Solving Satisfiability Problems Using Nondeterministic OBDDs

In this section, we analyze methods to solve satisfiability problems using OBDDs and observe that such methods can be regarded as using Nondeterministic OBDDs (NOBDDs).

Satisfiability problem of formulas or circuits is a fundamental problem. In order to solve satisfiability problems, it is sufficient to construct OBDDs for the formulas or circuits, because the reduced OBDD representing contradiction is unique. However, it is not necessary to construct complete OBDDs as long as we can check satisfiability easily. We can reduce required amount of storage at the cost of canonicity of OBDDs, by decomposing the function, or by introducing redundancy.

Recently, methods for manipulating Boolean functions without constructing whole OBDDs for given functions are used. Let us observe the easiest case first as an example. Let f be the function which we want to examine satisfiability. If f is decomposed into subfunctions as $f = f_1 \vee f_2 \vee \cdots \vee f_k$, we can examine the satisfiability of f by constructing OBDDs for $f_1, f_2, \ldots, f_k$ and checking satisfiability of each subfunction. When the size of OBDDs for each function is $s_1, s_2, \ldots, s_k$, the size of OBDD for f could be $O(s_1 \times s_2 \times \cdots \times s_k)$. If f is suitable for decomposition and the way of the decomposition is good, the OBDD size for $f_1, f_2, \ldots, f_k$ is much smaller than that of f.

Next, let us consider more general cases. Let $f(\vec{x})$ be a Boolean function. Then introduce new variables $\vec{y}$ in some manner and express $f(\vec{x})$ as $\hat{f}(\vec{x}, \vec{y})$ such that,

$$f(\vec{x}) \equiv \exists \vec{y}.\ \hat{f}(\vec{x}, \vec{y}).$$

Note that $f(\vec{x})$ is satisfiable if and only if $\hat{f}(\vec{x}, \vec{y})$ is satisfiable. The variable ordering for $\{\vec{x}, \vec{y}\}$ can be chosen arbitrary. On reduced OBDDs, satisfiability of $\hat{f}(\vec{x}, \vec{y})$, and hence satisfiability of $f(\vec{x})$, can be decided easily. We can use this property in solving satisfiability problems.

For some functions, the size of the OBDD for $\hat{f}(\vec{x}, \vec{y})$ is much smaller than that for $f(\vec{x})$, if $\vec{y}$ and ordering of $\{\vec{x}, \vec{y}\}$ are selected carefully.

Our question is as follows: what is the condition for such methods to work efficiently? Here, we define NOBDDs in order to formalize the expression in these methods and evaluate the expressive power.

Definition 3.1 An NOBDD is an OBDD whose variable nodes are labeled by either of n variables or an '$\vee$' symbol. While '$\vee$' symbols can appear arbitrary, appearance of the variables is restricted as usual OBDDs. Given an assignment, NOBDD is traversed nondeterministically on each node labeled by an '$\vee$' symbol. If at least one path reaches a constant node labeled by 1, the value of the function the NOBDD represents is determined to be 1, and otherwise, determined to be 0. □

The OBDD for $f(\vec{x}, \vec{y})$ can be regarded as an NOBDD for $f(\vec{x})$, if we regard the variable nodes labeled by variables in $\vec{y}$ as nodes labeled by '$\vee$.'

We consider how much the nondeterminism introduced here can improve the expressive power of OBDDs. Theorem 3.1 states the expressive power of OBDDs. We show the similar property for NOBDDs. The main idea is the same as in [24], where other stronger models are studied. Let *PolyNOBDD* be the class of functions representable by NOBDDs of polynomial size.

Theorem 3.2 $PolyNOBDD = \Pi(1\text{-}NL/poly)$

Proof: Let n be the input size. First, we simulate an NOBDD D by a $\Pi(1\text{-}NL/poly)$ Turing machine M. The description of the NOBDD for input size n (encoded in binary string) is given as the advice. The advice is a function of n and the length is polynomial in n. M starts at the root node of D and traverse the graph. On the nodes labeled by a variable, M reads the input value and traverses the corresponding edge. On the nodes labeled by '$\vee$,' M traverses both outgoing edges nondeterministically. $O(\log n)$ space is sufficient in order to store the current node number and to find the next node, as the description of D is given as the advice. In the computation, the order M reads the input bits is fixed for each input size.

Conversely, we simulate a $\Pi(1\text{-}NL/poly)$ Turing machine by an NOBDD. We assign a node of OBDD for each configuration where i) an input bit is read, or ii) a nondeterministic branch occurs. Each node corresponding to a read configuration (case i)) has the corresponding variable as its label. Each node corresponding to a nondeterministic branch (case ii)) has a '$\vee$' as its label. The edges are assigned according to the transition relation. The number of nodes is at most polynomial in n because there are at most polynomial number of configurations. The order in which variables appears is fixed for each input size. □

Similar discussion is possible for the case that conjunctive quantifiers are allowed, where the corresponding class is $\Pi(co\text{-}1\text{-}NL/poly)$.

Since $\Pi(1\text{-}L/poly) \subsetneq \Pi(1\text{-}NL/poly)$, we can say that introduction of '$\vee$' nodes properly boosts up the expressive power of OBDDs at the cost of canonicity.

However, in order to utilize the nondeterminism, heuristic methods for constructing NOBDDs, in particular for introducing '$\vee$' nodes and ordering variables, are required. In the next two sections, we show two particular cases where subclasses of NOBDDs are constructed by taking the structure of instances into account.

3.4 Combinational Circuits with Small Cutwidth

In this section, we show that, when combinational circuits are given, we can partly get a clue to minimize the NOBDDs of the functions the circuits compute, from the structure of the circuits.

It is well-known that the width of circuits correspond to the space of deterministic Turing machines[30, 12]. As OBDD size is also related to the space of Turing machines, OBDD size can be estimated from the width of the circuit which computes the function[4, 23].

Hamaguchi et al. showed a method for equivalence check of combinational circuits using OBDDs with redundant variables[18]. This method can be regarded as constructing NOBDDs for given circuits. In this method, the given circuit is partitioned into modules so that the modules are arranged in line and the interconnection wires exist only between adjacent modules. Then redundant variables are inserted and an OBDD with redundant variables is constructed from the OBDDs for each modules. The size of the resulting OBDD is related to the number of interconnection wires.

In this section, we evaluate the class of functions on which this method can work efficiently with respect to the OBDD size. We introduce circuit *cutwidth*, a measure similar to circuit width, and relate it to the space of Turing machines. Because the space complexity of nondeterministic online Turing machines corresponds with the size of NOBDDs as shown in Theorem 3.2, we can obtain a relationship between circuit cutwidth

and the size of NOBDDs.

In the definitions below, we treat circuits as directed acyclic graphs. Let $G = (V, E)$ be a directed graph, where V is a set of vertices and E is a set of edges. A linear arrangement of G is a bijection $L : V \to \{1, 2, \cdots, |V|\}$. The cutwidth of G is the minimum of

$$\max_{1 \le i < |V|} |\{(u, v) \in E \quad | \quad L(u) \le i < L(v) \quad \text{or} \quad L(v) \le i < L(u)\}|$$

for every linear arrangement L.

Definition 3.2 *1-LogCut* is the class of functions computed by families of polynomial size circuits with cutwidth $O(\log n)$, where n is the size of inputs.

Here, each circuit can have at most one input gate for each input bit. □

Note that the width of a circuit can be defined by restricting linear arrangement so that there are no edges (u, v) with $L(v) < L(u)$.

The next theorem states that any function in *PolyOBDD* is computed by a *1-LogCut* circuit family, but the converse is not true.

Theorem 3.3 $\Pi(\textit{1-L/poly}) \subsetneq \textit{1-LogCut}$

Proof: ($\subseteq$) It is known that *L/poly* corresponds to the class of functions computable by a family of circuits of $O(\log n)$ width. It is easy to show that $\Pi(\textit{1-L/poly})$ corresponds to the subclass with restriction that each circuit have at most one input gate for each input bit. Because the circuit cutwidth is less than or equal to the circuit width, the inclusion holds.

($\neq$) It is sufficient to show a function which is in *1-LogCut* but not in $\Pi(\textit{1-L/poly})$. The family of Hidden Weighted Bit functions $\{\mathrm{HWB}_n\}$ is known not to be in $\Pi(\textit{1-L/poly})$[6]. For input $x_1 x_2 \cdots x_n$, the weight $w(x_1 x_2 \cdots x_n)$ is defined as the number of '1's, that is,

$$w(x_1 x_2 \cdots x_n) = |\{i \mid x_i = 1\}|.$$

HWB_n is defined as :

$$\mathrm{HWB}_n(x_1 x_2 \cdots x_n) = \begin{cases} 0, & w(x_1 x_2 \cdots x_n) = 0 \\ x_{w(x_1 x_2 \cdots x_n)}, & \text{otherwise.} \end{cases}$$

As the circuit for HWB_n can be constructed from a parallel adder, which computes w, and a multiplexer tree, which selects x_w, it is easy to check that $\{\mathrm{HWB}_n\}$ is in *1-LogCut*. □

We show in the next theorem that any function computed by a *1-LogCut* circuit family is in *PolyNOBDD*, but the converse is not true.

Theorem 3.4 $\textit{1-LogCut} \subseteq \Pi(\textit{1-NL/poly} \cap \textit{co-1-NL/poly})$

Proof: Let us consider a circuit C and its arrangement L satisfying the conditions of *1-LogCut*. Let $G = (V, E)$ be the directed graph underlying C. We can assume that the output gate of C is in the rightmost ($|V|$'th) place in L, without loss of generality. (If it is not the case, we can fold the arrangement and construct an arrangement with cutwidth at most twice.)

In Figure 3.2(a), vertices (gates) represented by boxes are arranged in line. For each edge directed to the left, insert an input gate of dummy variable, as shown in Figure 3.2(b). The dummy input is fed to the destination of original edge, and a comparator, which outputs '1' when the value is equal to the value of the source of the original edge. The outputs of comparators are and'ed with the original output to make new output. There exists an assignment to the dummy variables with which this new output has the value 1, if and only if the original function has the value 1.

The new circuit is arranged so that all the edges are directed to the right. We can simulate this circuit by a $\Pi(\textit{1-NL/poly})$ machine in the similar way as [30, 12]. The only difference we have to note is that there are dummy variables, which can be simulated by guessing the value nondeterministically. Therefore, $\textit{1-LogCut} \subseteq \Pi(\textit{1-NL/poly})$.

Because *1-LogCut* is closed under complement, we can also say that $\textit{1-LogCut} \subseteq \Pi(\textit{co-1-NL/poly})$. □

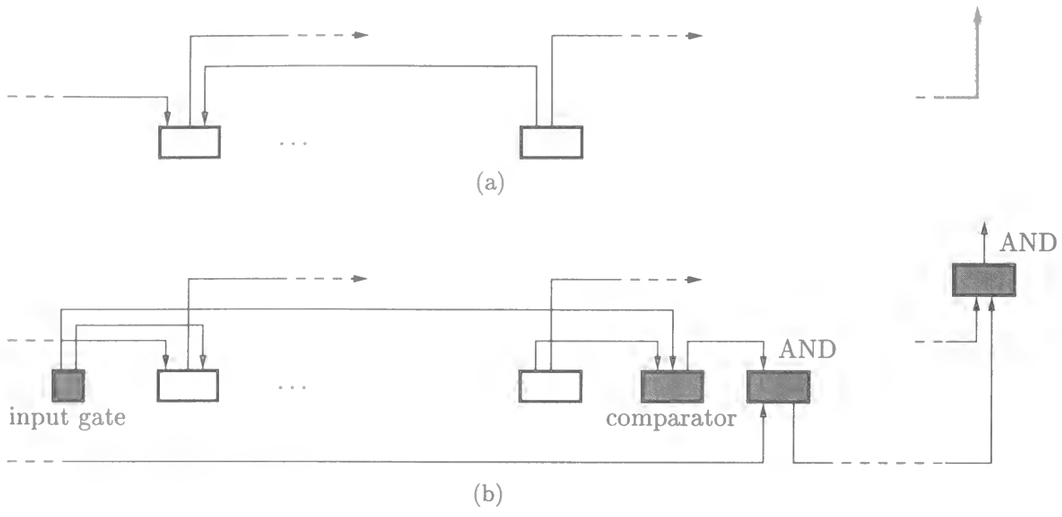


Figure 3.2: Replacing two-way cuts with one-way cuts.

Noting the fact that $\Pi(\textit{1-NL/poly}) \neq \Pi(\textit{co-1-NL/poly})$, the next corollary is derived.

Corollary 3.1 *1-LogCut* $\subsetneq \Pi(\textit{1-NL/poly})$ (or, $\Pi(\textit{co-1-NL/poly})$) □

This theorem leads to a natural extension of known relationship between circuit width and OBDD size[4, 23]. However, as shown above, cutwidth does not correspond exactly to the NOBDD size.

3.5 OBDDs Representing Sum-of-Products Form

Coudert et al.[11] developed methods for two-level logic minimization using Meta Products, which represent set of product terms using OBDDs. Minato[25] have proposed zero-suppressed BDDs (0-Sup-BDDs), a variant of reduced OBDDs with a reduction rule suitable for representing a set of combinations. A Zero-suppressed BDD can be used to represent a set of terms because terms can be regarded as combinations of literals.

Yasuoka[45] proposed a data structure called Ternary Decision Diagrams (TDDs), which are specialized for manipulation of sets of terms.

A set of product terms can be evaluated as a sum-of-products form. The three data structures above are essentially equivalent with respect to the expressive power in polynomial size, when we regard them as representations of Boolean functions through sum-of-products form[27]. Let us regard these data structures as representations of functions through sum-of-products form and call 'OBDDsop.' In this section, we evaluate the expressive power of OBDDsop for Boolean functions.

For a variable x_i and a term p, there are three cases of occurrence of literals; x_i occurs in p, $\overline{x_i}$ occurs in p, and neither x_i nor $\overline{x_i}$ occurs in p. In [11] and [25], a set of terms is represented using two new variables for each x_i to distinguish these three cases on OBDDs. In [25], a term that consists of n variables $x_1, x_2, \ldots, x_n$ is represented by a $2n$-bit vector $(x_1\overline{x_1}x_2 \cdots x_n\overline{x_n})$, where each bit, x_i or $\overline{x_i}$, expresses whether the corresponding literal is included in the term or not. The variables are ordered so that x_i and $\overline{x_i}$ are adjacent for all i. For example, a product term $x_1\overline{x_2}x_4$ can be represented by (10010010). [11] uses a different encoding, but it is essentially equivalent to [25]. Given a set P of product terms, the function $f_P(x_1, \overline{x_1}, x_2, \overline{x_2}, \ldots, x_n, \overline{x_n})$ is defined such that $f_P(x_1, \overline{x_1}, x_2, \overline{x_2}, \ldots, x_n, \overline{x_n}) = 1$ if and only if the product term corresponding to the vector $(x_1\overline{x_1}x_2 \cdots x_n\overline{x_n})$ is in P. An OBDD is said to represent the set of product terms P if the OBDD represents the function $f_P(x_1, \overline{x_1}, x_2, \overline{x_2}, \ldots, x_n, \overline{x_n})$. In this expression, for each variable x_i, the nodes labeled by the positive literal x_i and the nodes labeled by the negative literal $\overline{x_i}$ are used in a OBDD.

The expression in TDDs can be constructed by collapsing each two adjacent levels into one. A TDD is similar to an OBDD except that each variable node has exactly three outgoing edges, 0-edge, 1-edge, and $*$-edge, and a variable can take these three values, where 1 (or 0) means that the positive (or negative) literal occurs, and $*$ means that the literal does not occur. Thus the occurrence of a literal is expressed by a node. A TDD representing the set $\{x_1\overline{x_2}x_3, \overline{x_1}\,\overline{x_3}, x_2\overline{x_3}\}$ is shown in Figure 3.3.

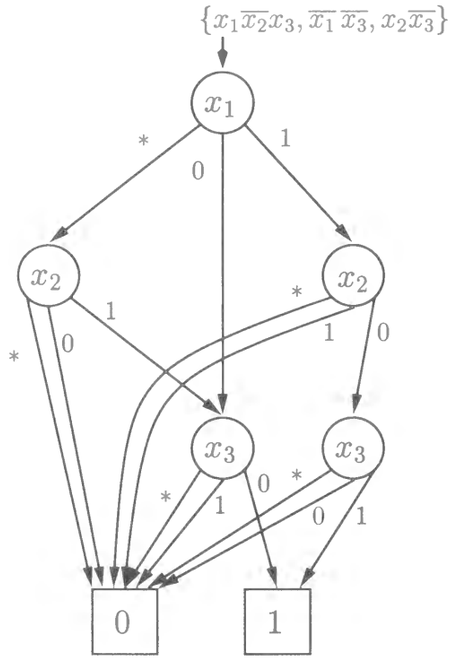


Figure 3.3: A TDD representing $\{x_1\overline{x_2}x_3, \overline{x_1}\,\overline{x_3}, x_2\overline{x_3}\}$.

We investigate here the expressive power of these data structures as representations of functions. Let *PolyOBDDsop* be the class of functions computable by polynomial size OBDDsop. The next theorem states that the class *PolyOBDDsop* strictly includes *PolyOBDD*.

Theorem 3.5 $\Pi(\textit{1-L/poly}) \subsetneq \textit{PolyOBDDsop}$

Proof: ($\subseteq$) Consider the TDD D representing the set of minterms of given function f. We can construct the OBDD representing f of the same size by removing *-edges.

($\neq$) Let $G = (V, E)$ be an m-dimensional hypercube, with $V = \{0, 1, \ldots, 2^m - 1\}$ and $E = \cup_{i=0,1,\ldots,m-1} E_i$, where $E_i = \{\{u, v\} \mid u - v = 2^i\}$.

Let us define the following functions of variables $a_0, \ldots, a_{m-1}, x_0, \ldots, x_{2^m-1}$, and let $n = m + 2^m$.

$$f_n = \bigvee_{i=0}^{m-1} a_i g_i$$
$$g_i = \bigvee_{\{u,v\} \in E_i} x_u \wedge x_v$$

The number of terms is polynomial in n and therefore $\{f_n\}$ is in *PolyOBDDsop*.

We now show that $\{f_n\} \notin \Pi(\textit{1-L/poly})$. Consider each g_i and take arbitrary permutation $\pi_{2^m} : \{0, \ldots, 2^m - 1\} \to \{0, \ldots, 2^m - 1\}$.

Let $X_L = \{x_{\pi(0)}, \ldots, x_{\pi(2^{m-1}-1)}\}$ and $X_R = \{x_{\pi(2^{m-1})}, \ldots, x_{\pi(2^m-1)}\}$. (X_L, X_R) corresponds to a bisection of the set of vertices V. Let $\hat{E}_i = \{\{u, v\} \mid \{u, v\} \in E_i$ and $x_u \in X_L$ and $x_v \in X_R\}$. Because the bisection width of a m-dimensional hypercube is not less than 2^{m-1}[22], for some i, $|\hat{E}_i| \geq \frac{2^{m-1}}{m}$. Let i_m be an i satisfying this condition and let $\hat{g}_{i_m} = \vee_{\{u,v\} \in \hat{E_{i_m}}} x_u \wedge x_v$.

Note that we can yield $\hat{g}_{i_m}$ from f_n, by assigning 1 to a_{i_m} and 0 to all $a_j, j \neq i_m$ and 0 to all x_j that do not appear in $\hat{g}_{i_m}$. Therefore the computation space required for f_n is not less than that of $\hat{g}_{i_m}$.

Now let us suppose that $\{f_n\} \in \Pi(\textit{1-L/poly})$. Then, $\{f_n\}$ can be computed by an *1-L/poly* machine after some input permutation which is fixed for each input length.

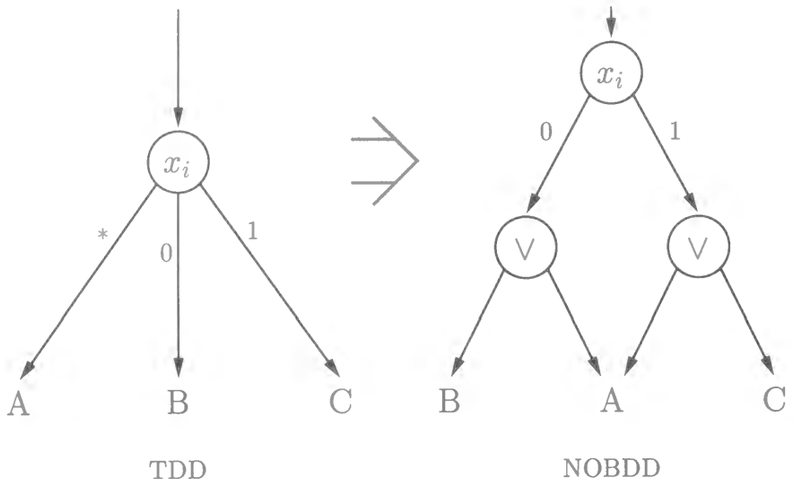


Figure 3.4: Converting TDD to NOBDD.

Therefore, $\hat{g}_{i_m}$ is also computed by an *1-L/poly* machine after input permutation $\{\pi_{2^m}\}$. (Recall that the input length of $\hat{g}_{i_m}$ is polynomially related to n.) Let M be the *1-L/poly* machine.

The number of possible configurations of M is bounded by polynomial in n because of the space bound. However, assigning constants for variables in X_L yields at least $2^{|\hat{E_{i_m}}|}$ different functions from $\hat{g}_{i_m}$. This is a contradiction.

□

An OBDDsop can be regarded as a NOBDD of restricted form.

Theorem 3.6 *PolyOBDDsop* $\subseteq \Pi$(*1-NL/poly*)

Proof: When a TDD is given, we can construct an NOBDD, that represents the same function as the TDD, with asymptotically the same size. For each node v in the given TDD, we replace v with a graph as shown in Figure 3.4. □

We show that *PolyBDDsop* is strong enough that a complete language for *1-NL* belongs

to *PolyBDDsop*. However, whether *PolyBDDsop* is equal to Π(*1-NL/poly*) or not is not known.

Let $G = (V, E)$ be a graph and $V = \{v_1, v_2, \ldots, v_m\}$. The characteristic function of language TAGAP is the family of functions $\{\text{TAGAP}_m\}$ defined as follows.

> $\text{TAGAP}_m(x_{12} \cdots x_{1m} x_{21} \cdots x_{2m} \cdots x_{mm}) = 1$ iff (x_{ij}) is the adjacency matrix of a directed acyclic graph G such that G is topologically arranged and there exists a path from v_1 to v_m in G.

Here, the representation of a directed acyclic graph G is topologically arranged if there is no edge (v_i, v_j) when $i > j$. It is known that TAGAP is a complete language for *1-NL* under *1-L* reductions.

Theorem 3.7 TAGAP $\in$ *PolyOBDDsop*

Proof: Let f_k be the function that $f_k = 1$ if and only if there exists a path from v_k to v_m in G. $f_1 = \text{TAGAP}_m$ by the definition. We recursively construct a TDD for f_{m-i} , for $i = 1, \ldots, m-1$, as

$$\begin{aligned} f_{m-i} &= (x_{m-i\ m-i+1} \wedge f_{m-i+1}) \vee \\ &\quad (x_{m-i\ m-i+2} \wedge f_{m-i+2}) \vee \cdots \vee x_{m-i\ m}. \end{aligned}$$

The size of the TDD increases by i and therefore the size of $f_1 = \text{TAGAP}_m$ is $O(m^2)$. The resulting TDD when $m = 5$ is shown in Figure 3.5. In this figure, all the 0-edges are omitted. The 0-edge from a node v points to the same node as the $*$-edge from v, or points to the constant 0. □

3.6 Conclusion

In this chapter, we introduced Nondeterministic OBDDs and observed that some applications of OBDDs can be viewed as utilizing the power of nondeterminism in order to reduce the size of OBDDs.

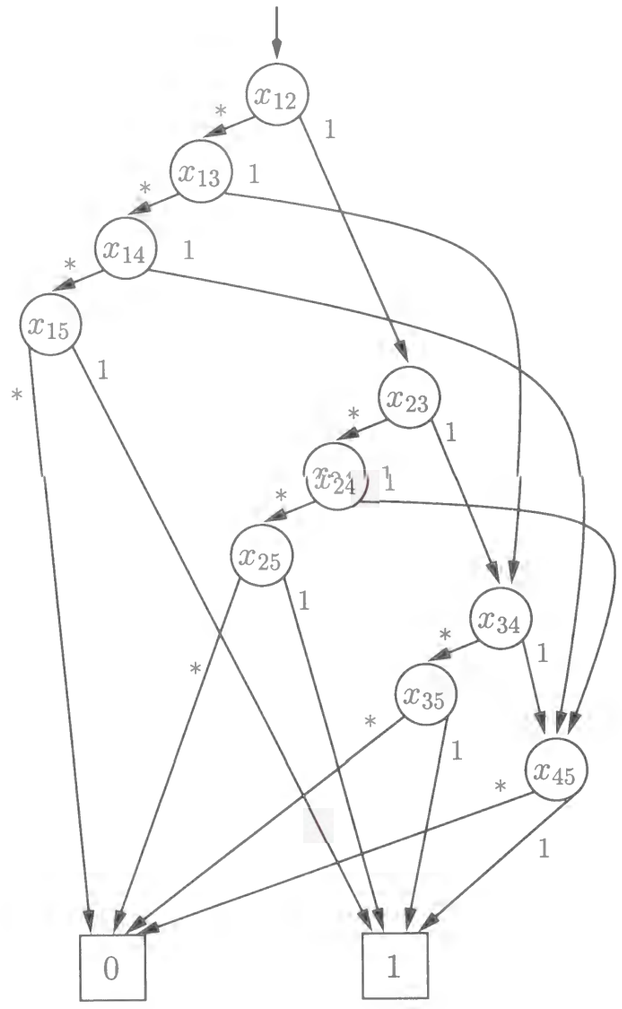


Figure 3.5: A TDD representing TAGAP ($m = 5$).

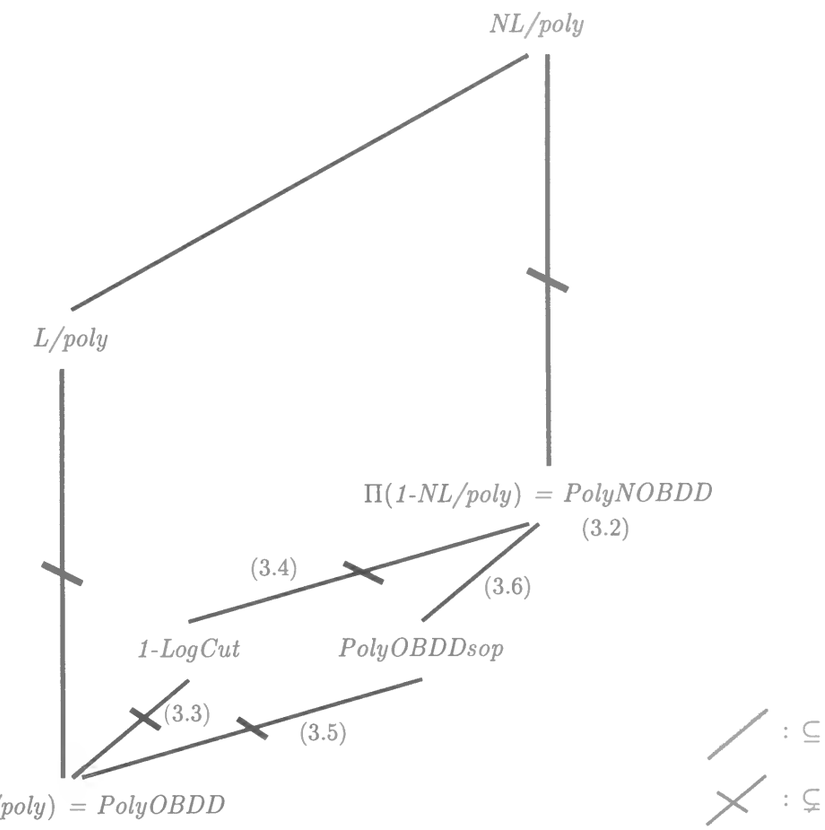


Figure 3.6: Relationships among classes of functions.

We showed that the cutwidth of combinational circuits is related to the size of NOBDDs, so, methods for circuit partitioning can be applied to the reduction of NOBDD size. We also showed that OBDDs representing sum-of-product form can be regarded as a restricted form of NOBDDs. We summarize the relationships among classes of functions in Figure 3.6. A line represents that the above class contains the below class. A line with a check represents proper containment. The relationships proved in this thesis are accompanied with the theorem numbers.

It is possible to consider OBDDs with universal quantifiers, or parity quantifiers, in the same way. When universal quantifiers are introduced, the corresponding class is $\Pi(co\text{-}1\text{-}NL/poly)$, and the corresponding formula is product-of-sum form.

Chapter 4

Exact Minimization of Free Binary Decision Diagrams for Pass-Transistor Logic Optimization

4.1 Introduction

With the advances in fabrication technology, low power design of VLSI has become a matter of great importance. Alternatives to the conventional CMOS design style have been sought extensively in order to reduce the power dissipation. Recently, Pass-Transistor Logic(PTL) circuits have been paid attention for the potential of low-power and high-speed circuit implementation compared to CMOS circuits.

Pass-transistors have been used as transfer gates since the age of early NMOS logic. PTL discussed here is a form of pass-transistor networks where each transistor makes a pair with its complement. A transistor pair works as a selector and hence the whole circuit works as a static selector logic circuit. A variety of PTL circuit structures have been proposed such as Complementary Pass-transistor Logic (CPL)[44], Double Pass-transistor Logic (DPL)[36], Swing Restored Pass-transistor Logic (SRPL)[29], Lean Integration with Pass-transistors (LEAP)[43] and Single-rail Pass-transistor Logic (SPL)[38]. Because a

function requiring several gates in CMOS can be implemented in a single selector network in PTL, the area and power dissipation could be reduced with PTL. Though there is room for further investigation, PTL is regarded as a promising alternative to CMOS.

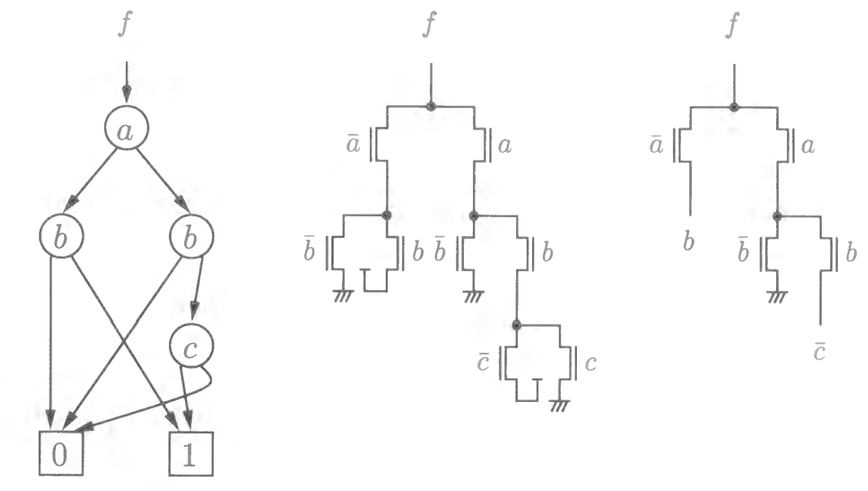
Several attempts have been done to establish design methodology for PTL circuits. [43] and [8] introduced synthesis flow for PTL based on Ordered Binary Decision Diagrams (OBDDs)[1, 5]. In their synthesis flow, Boolean functions are decomposed and expressed as aggregate of small OBDDs, and the component OBDDs are mapped directly to PTL cells. Their experimental results show that PTL could outperform CMOS. However, they also suggest that the synthesis and optimization methods for PTL circuits are not yet mature.

In the synthesis methods for PTL based on OBDDs in decomposed form, the size of decomposed OBDDs directly affects the circuit size. Therefore, minimization of decomposed OBDD is one of the main goals. We can utilize some extension of OBDDs instead of conventional OBDDs for further minimization of the function expression. In this chapter, we investigate a method using Free BDDs (FBDDs)[16], which are well-studied extension of OBDDs.

A heuristic method for minimization of FBDDs has been shown in [37]. There proposed several algorithms based on simulated annealing, in which, functions are once expressed in OBDDs and the variables on some paths are reordered to construct FBDDs and then minimization is performed. Their experimental results show that the algorithms obtain smaller expression size than the initial OBDDs for benchmark circuits. However, it is not known how their results compare with the optimum OBDDs or the optimum FBDDs.

This chapter aims at estimating the gain of employing decomposed FBDDs instead of decomposed OBDDs. First, we show statistics of the size of minimum OBDDs and minimum FBDDs. We also show experimental results on the size of decomposed FBDDs of benchmark circuits based on exact minimization of the component FBDDs.

In the next section, we review BDD-based PTL synthesis flow and present our motivation for exact minimization of FBDDs. The problem of minimizing FBDDs is analyzed in



(a) OBDD (b) PTL (c) PTL without pre-terminal nodes

Figure 4.1: Synthesis of PTL circuit from OBDD.

Section 3. In Section 4, we show statistics of the sizes of OBDDs and FBDDs. Section 5 contains the experimental results on benchmark circuits. Section 6 is a conclusion.

4.2 Decomposed BDDs and Pass-Transistor Logic Synthesis

The synthesis methods for PTL in [43] and [8] are based on the construction of OBDDs. Once a Boolean function is expressed as an OBDD, the PTL circuit is derived directly. Fig. 4.1(a) is an example of an OBDD for a function with three input variables, $f = \bar{a}b + b\bar{c}$. The edges from the left and right of each node are 0-edge and 1-edge respectively. PTL circuit can be obtained by replacing each node of the OBDD by a pair of transistors gated by the control signal and its negation, as shown in Fig. 4.1(b).

Circuit design as shown in Fig. 4.1(c) is possible, where the pre-terminal nodes, the

nodes whose successors are both constant nodes, are suppressed by connecting the corresponding signal to the drain. In the following, we mainly use this construction.

The size of PTL circuits is proportional to the size of OBDDs. Therefore, minimization of OBDD size results in minimization of PTL circuit size. In this chapter, we deal with the size of decision graphs including and excluding pre-terminal nodes. Note that the total transistor count and area, including buffers and wires, depends on detailed design styles, but it is beyond the scope of this chapter.

In PTL circuits, the length of pass-transistor chain is bounded since long pass-transistor chain leads to poor performance. In OBDD-based synthesis, the chain length can be at most the number of the input variables. Therefore, the number of input variables of OBDDs is usually limited.

In the synthesis flow of [43], monolithic OBDDs are built and then buffers are inserted to cut the chains into chunks. [8] takes another approach which constructs decomposed OBDDs without dealing with monolithic OBDDs. In their algorithm, a set of OBDDs are constructed for a function in a depth-first manner from inputs to outputs and decomposition points are selected during the construction in order to control the size of the individual OBDDs.

As shown above, the synthesis of PTL circuits has two phases: i) decomposition into subfunctions with limited number of inputs, and ii) expression of the individual components. In this chapter, we focus on the latter problem. On OBDDs, the choices of the expressions of functions only come from the variable ordering. However, there is a possibility of expression with less size than OBDDs by relaxing the restriction of the variable occurrence on OBDDs at the expense of the canonicity. As far as PTL synthesis is concerned, general decision graphs can be used.

In this chapter, we employ FBDDs[16] as the expression of the component functions instead of OBDDs, and consider their exact minimization. On FBDDs, each variable can appear at most once on each path, but the order can be different among paths.

As the first step of using general decision graphs for PTL synthesis, FBDDs have

good properties to begin with. First, the number of variable ordering of an FBDD is bounded and the exact minimization is feasible for small number of variables. Secondly, the maximum path length is bounded by the number of variables, which is the same as OBDDs. From the next section, we evaluate the advantage of employing decomposed FBDDs instead of decomposed OBDDs with respect to the expression size.

4.3 Exact Minimization of FBDDs

Because the variable order of FBDDs can be different among paths, a set of orders can be expressed as labels on a complete binary tree. Figure 4.2 shows all the 12 orders for FBDDs with 3 variables a, b and c. Usually, these orders are expressed as FBDD types[16], which can be obtained by sharing isomorphic subgraphs of each tree. Note that the sharing may reduce the graph size, but do not reduce the number of possible combination of orders. In this chapter, we use the binary tree expression. This is because our algorithms include only the construction of minimum FBDDs and do not include operations on FBDDs. In such cases, there is little advantage of using FBDD types and rather the overhead for sharing would have to be taken account. Another reason is that it is feasible for our case of $n \leq 5$.

Let S_n be the number of possible variable orders for FBDDs of Boolean functions with n variables. When the variable for the root node is fixed, deciding the variable order of the rest is equivalent to deciding the variable orders of two $(n-1)$-variable FBDDs with the two successors of the root node as root nodes (see Figure 4.3.) This relation is written as follows:

$$\begin{aligned} S_n &= nS_{n-1}^2 \\ &= \prod_{k=1}^{n} k^{2^{n-k}}. \end{aligned}$$

The value of $n!$ (the number of all variable orders of OBDDs) and S_n for $n \leq 7$ is shown in Table 4.1.

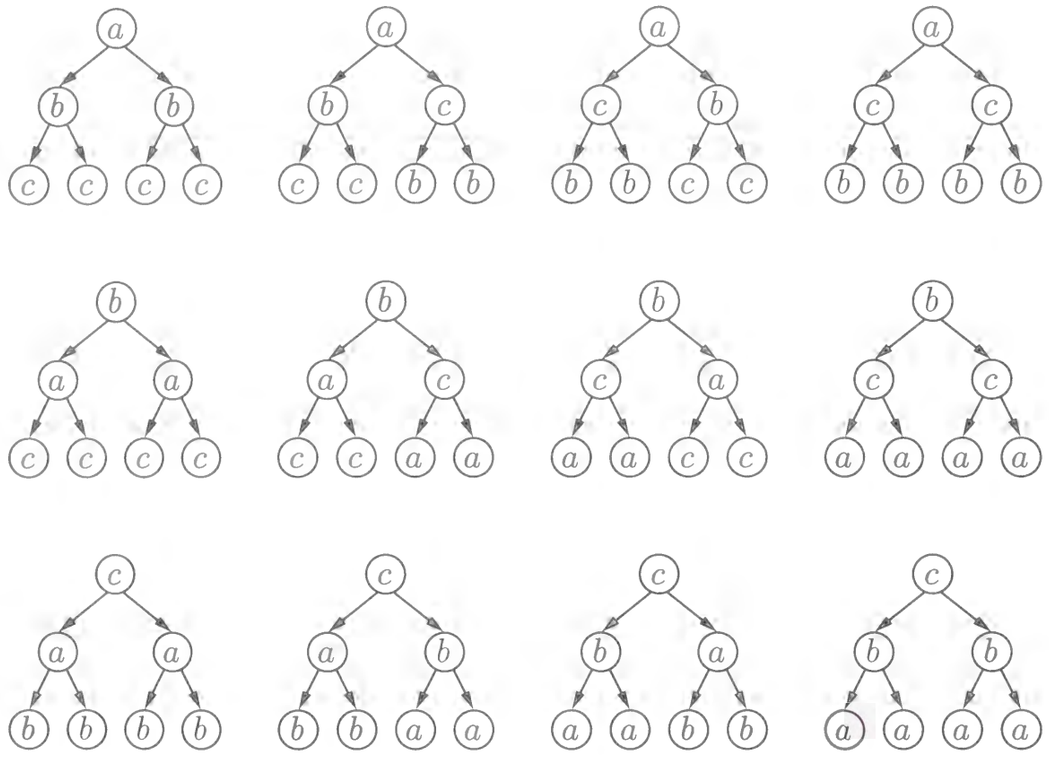


Figure 4.2: All variable orders of FBDDs for $n = 3$.

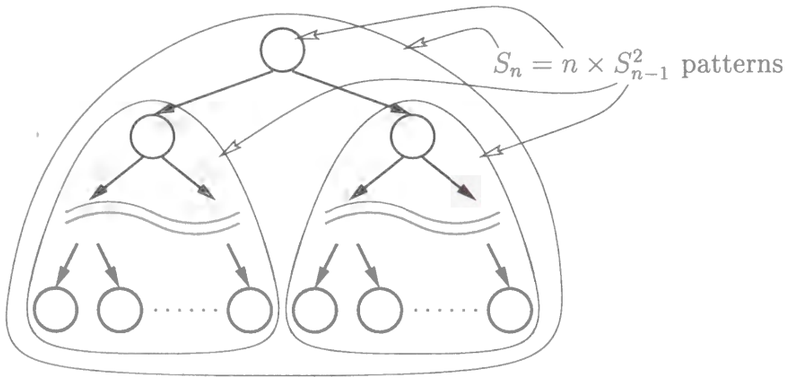


Figure 4.3: The number of FBDD variable orders.

Table 4.1: The number of variable orders of OBDDs and FBDDs.

n	OBDD ($n!$)	FBDD (S_n)
1	1	1
2	2	2
3	6	12
4	24	576
5	120	1658880
6	720	16511297126400
7	5040	1908360529573854283038720000

For exact minimization of FBDDs, the dynamic programming algorithm which can be used for OBDD minimization[13] does not seem to work. Currently, our algorithm checks all FBDD variable orders in sequence. For given truth table (Fig. 4.4 (a)) and an FBDD variable order (Fig. 4.4 (b)), we construct FBDD (Fig. 4.4 (c)) in a bottom-up manner. In order to find a minimum FBDD, we enumerate all possible FBDD variable orders in lexicographical order (e.g. row-first order in Fig. 4.2.)

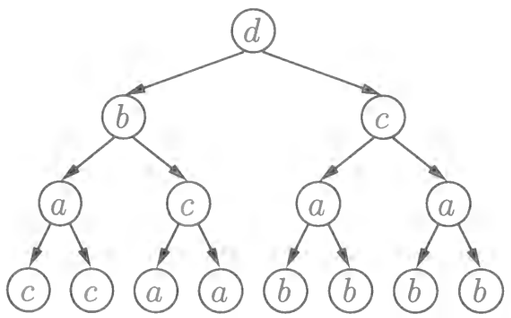
Several strategies for pruning the search space can be used. Let f be a function and T be a tree representing a variable order. Let us consider the following two cases.

- For a subtree T' of T, the subfunction of f corresponding to T' does not depend on all the variables on T'.
- For two subtrees T', T'' of T, where the sets of variables on T' and T'' are the same, the subfunctions of f corresponding to T' and T'' are identical.

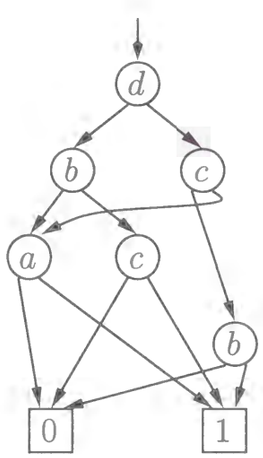
In the first case, the order, within T', of the variables which the subfunction does not depend is unrelated to the FBDD. Especially, if the subfunction is tautology or contradiction, any changes of order within T' result into the same FBDD as in original T. In the second case, it is safe to assume that T' is equal to T'', because FBDDs for the two sub-

a	b	c	d	f	a	b	c	d	f
0	0	0	0	0	1	0	0	0	1
0	0	0	1	0	1	0	0	1	0
0	0	1	0	0	1	0	1	0	1
0	0	1	1	0	1	0	1	1	1
0	1	0	0	0	1	1	0	0	0
0	1	0	1	1	1	1	0	1	1
0	1	1	0	1	1	1	1	0	1
0	1	1	1	0	1	1	1	1	1

(a) truth table



(b) variable order



(c) FBDD

Figure 4.4: Construction of FBDD.

functions can be shared without requiring extra nodes. We can skip some of the variable orders not necessary for FBDD minimization by checking these cases.

4.4 Minimum Size of OBDDs and FBDDs

n-variable Boolean functions f and g are said to be NPN-equivalent[26] when g is constructed from f by combination of the following three types of operations: i) negation of some of the input variables of f, ii) permutation of input variables of f, and, iii) negation of f. NPN-equivalence relation induces equivalence classes of Boolean functions. We minimized FBDDs for all the NPN-representative functions for number of variables n as follows.

For each f : NPN-representative of 2^{2^n} functions,

1. Find an OBDD of f with minimum size.
2. Find an FBDD of f with minimum size.

Because the minimum size of OBDDs (or FBDDs) for NPN-equivalent functions are the same, it is possible to obtain statistics for all functions from this result.

The algorithm for FBDD minimization is basically an exhaustive search with the pruning idea presented in the previous section taken into consideration. The next proposition is useful to suppress searching on functions which have no possibility of improvement.

Proposition 4.1 Let f be a function which depends on exactly n variables. If the size of an FBDD for f is equal to n, the FBDD can be regarded as an OBDD with some variable order.

Proof) Since f depends on n variables, there should be at least one node, and hence exactly one node, for each variable. The variable order as an OBDD is obtained from a topological order of the FBDD nodes. □

Corollary 4.1 Let f be a function which depends on exactly n-variables. If the minimum size of OBDDs for f is $n+1$, the minimum size of FBDDs for f is also $n+1$. □

The FBDD minimization algorithm is implemented in C language. The experiments are carried out on AlphaServer8400 (DEC alpha21164A CPU 617MHz × 10, 8GB main storage).

Table 4.2 shows statistics of the minimum size of OBDDs and FBDDs for all functions of 4 variables. The 222 NPN-equivalence classes are partitioned into groups according to the minimum OBDD-size. For each group, FBDD-size of the functions in the group is shown. The '#class' and '#function' columns show the number of NPN-equivalence classes and the number of functions included in the classes respectively. FBDD is smaller than OBDD for 13 classes among the 222 classes. The CPU time for FBDD minimization for all 222 representative functions was about 1.8 seconds.

Table 4.2: Minimum size of OBDDs and FBDDs ($n = 4$).

OBDD size	#class	#function	FBDD size	#class	#function
0	1	2	0	1	2
1	1	8	1	1	8
2	1	48	2	1	48
3	4	364	3	4	364
4	14	3168	4	14	3168
5	38	12440	5	38	12440
6	70	22488	6	70	22488
7	68	20346	6	3	1536
			7	65	18810
8	25	6672	7	10	4032
			8	15	2640

In order to take the design as shown in Fig. 4.1(c) into account, we counted the nodes of OBDDs and FBDDs in another way, i.e., excluding pre-terminal nodes. In this case, we cannot use Proposition 4.1. The statistics for all 4-variable NPN-equivalence classes are shown in Table 4.3. The size decreased for 17 of the 222 classes. The CPU time is the same as the previous case.

Table 4.4 shows the results for 5-variable functions. Among 616126 NPN-equivalence classes, the size decreased for 345073 classes. The CPU time for FBDD minimization for a 5-variable function was about 15 seconds. Up to 40MB of main storage was used.

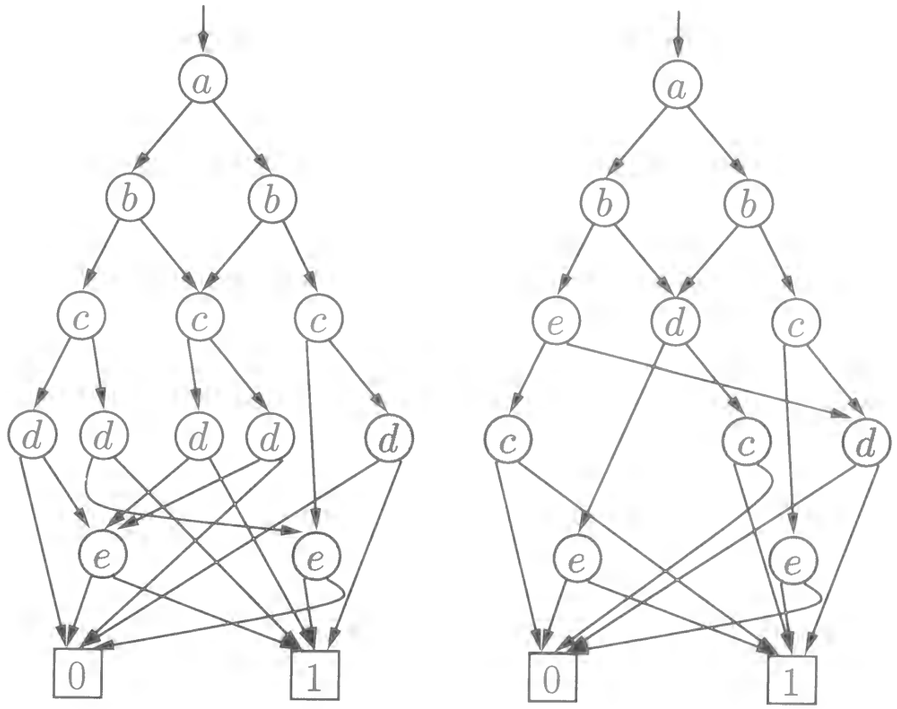
A function with minimum OBDD-size 10 and minimum FBDD-size 6 (excluding pre-terminal nodes) is shown in Fig. 4.5. Shadowed nodes are pre-terminal nodes. The hexadecimal expression of the truth table of this function is `167e8699`.

Table 4.3: Minimum size of OBDDs and FBDDs excluding pre-terminal nodes ($n = 4$).

OBDD size	#class	#function	FBDD size	#class	#function
0	2	10	0	2	10
1	3	156	1	3	156
2	11	2464	2	11	2464
3	43	12912	3	43	12912
4	72	24248	4	72	24248
5	77	23650	3	4	1344
			4	10	5760
			5	63	16546
6	14	2096	5	3	960
			6	11	1136

Table 4.4: Minimum size of OBDDs and FBDDs excluding pre-terminal nodes ($n = 5$).

OBDD size	#class	#function	FBDD size	#class	#function
0	2	12	0	2	12
1	3	340	1	3	340
2	12	12640	2	12	12640
3	107	301460	3	107	301460
4	724	3322584	4	724	3322584
5	4133	23295290	3	6	14400
			4	78	543360
			5	4049	22737530
6	18536	114688440	4	107	693120
			5	1646	11736000
			6	16783	102259320
7	64757	424248610	5	1065	7416640
			6	14499	105483328
			7	49193	311348642
8	172803	1185295712	5	446	3221760
			6	9994	73603840
			7	63754	464248128
			8	98609	644221984
9	228594	1618747984	5	6	23040
			6	1677	12528960
			7	33832	250282080
			8	109063	796088384
			9	84016	559825520
10	108673	795031200	6	42	288000
			7	2240	16705920
			8	33386	250655040
			9	55837	411282112
			10	17168	116100128
11	17543	128650656	7	142	909120
			8	1849	13777920
			9	9874	73436640
			10	5292	38157312
			11	386	2369664
12	239	1372368	8	14	103680
			9	84	478944
			10	112	685440
			11	28	104064
			12	1	240



(a) OBDD (b) FBDD

Figure 4.5: A function with OBDD-size 10 and FBDD-size 6 (excluding pre-terminal nodes.)

4.5 Experimental Results on Benchmark Circuits

We applied our FBDD minimization method for PTL synthesis to MCNC benchmarks as follows.

For each circuit computing Boolean function f,

1. Decompose f into 5-variable subfunctions $f_0, \ldots, f_m$.
2. For each subfunction f_i,
   (a) Find an OBDD of f_i with minimum size.
   (b) Find an FBDD of f_i with minimum size.

Since we have not yet developed function decomposition algorithms, we used Look-Up Table mapping algorithm of [32] for decomposition in the step 1. Each benchmark circuit is decomposed by mapping to 5-1 Look-Up Table cells. Then OBDDs and FBDDs for the subfunctions are minimized in the step 2. and the sizes are summed up. The results are shown in Table 4.5. The columns '#input' and '#cells' are the number of primary inputs and the number of mapped 5-1 cells respectively. Total number of nodes (excluding pre-terminal nodes) of OBDDs and FBDDs are shown in the next columns. The CPU time in seconds for minimization (step 2) is shown in the last column. This does not include the time for decomposition (step 1.) This CPU time is the sum of the time for minimization of both OBDDs and FBDDs, though the time for OBDDs is much smaller than that for FBDDs and can be neglected. 7 out of 16 circuits are reduced by using FBDDs.

We also applied our minimization method for ISCAS85 benchmarks. We used SIS for mapping the circuits into 5-1 Look-Up Tables. The results are shown in Table 4.6. Only one circuit is reduced by using FBDDs.

As we have noted, the total size of decomposed BDDs depends on both the decomposition method of the original function and the expression method of the individual

Table 4.5: Minimum OBDD and FBDD size of MCNC Benchmark Circuits.

	#input	#cell	OBDD size	FBDD size	time (sec.)
5xpl	7	21	41	41	95
9sym	9	7	42	42	34
alu4	14	183	1084	1027	1170
apex5	117	329	852	842	293
b12	15	25	53	53	51
clip	9	17	60	58	38
cordic	23	10	39	39	9
misex1	8	17	45	44	47
misex2	25	54	111	111	23
misex3	14	241	1419	1352	1292
misex3c	14	81	379	359	395
rd73	7	9	31	31	42
rd84	8	11	46	46	41
sao2	10	25	112	109	82
t481	16	6	19	19	3
vg2	25	29	97	97	49

Table 4.6: Minimum OBDD and FBDD size of ISCAS85 Benchmark Circuits.

	#input	#cell	OBDD size	FBDD size	time (sec.)
c432	36	66	214	214	37
c499	41	65	282	282	115
c880	60	89	309	309	99
c1355	41	65	282	282	280
c1908	33	103	368	368	284
c2670	233	253	534	534	272
c3540	50	382	1270	1270	372
c5315	178	411	1101	1100	918
c6288	32	770	2634	2634	127
c7552	207	512	1447	1447	1136

subfunctions. Though we focused on the latter process in this chapter, the effect of the former process might be heavier. The algorithms we used here are for Look-Up Table mapping, so, the cost function for decomposition may not match for our purpose. Nevertheless the result in Table 4.5 suggests that using FBDD would be promising.

The poor results in Table 4.6 might be due to the decomposition method. Because the original benchmark circuits are mostly composed of simple gates, their decomposition by SIS based on circuit covering and simplification results in rather simple 5-1 subfunctions. FBDD is not effective in such cases because OBDDs for subfunctions are too small. This is in contrast to the result in Table 4.5, where the whole function is once represented as an OBDD and then decomposed by Boolean function manipulation.

It is interesting how the processes for function decomposition and subfunction expression have influence on each other. Properties of the subfunctions depends on the decomposition algorithms and effect of using FBDD largely depends on the properties. Roughly speaking, FBDD will be effective when each component function has enough large OBDD. This implies that FBDD expression might not be useful when the circuits are once collapsed into two-level circuits and then decomposed, because such circuits could made up mostly of simple subfunctions. Methods based on function decomposition would be suitable if the number of subfunctions can be reduced by cramming much logic into each subfunction. However, such methods cannot treat large scale circuits in general.

4.6 Conclusion

In this chapter, we showed statistics of the size of minimum OBDDs and minimum FBDDs for all functions of fixed number of variables. The results can be used for PTL synthesis as libraries. We also applied the exact minimization algorithm of FBDDs to benchmark circuits. The results would encourage employing decomposed FBDDs instead of decomposed OBDDs for PTL synthesis.

The FBDD minimization algorithm shown here cannot deal with larger number of

variables. Though the problem seems to be inherently difficult, more pruning strategies could be introduced. We used FBDD as an extension of OBDDs, but more generic decision graphs could be employed. The most important point to put our results into practical use is development of decomposition algorithms suitable for FBDDs. This is also a future work.

Chapter 5

Timing Analysis of Sequential Logic Circuits with Multi-Clock Operations

5.1 Introduction

The clock frequency of a sequential logic circuit is decided based on the maximum delay of the combinational parts of the circuit. The precise estimation of the maximum delay is important in deciding the proper clock frequency. The maximum delay can be computed as the longest path of the weighted graph corresponding to the circuit, where nodes in the graph are logic gates in the circuit and the weight of the node is the delay time of each gate. The computational complexity of the maximum delay computation is proportional to the size of the graph.

In some cases, such topological maximum delay paths cannot be sensitized with any input patterns and therefore become false paths. Many researches have been done to detect false paths and to minimize the clock period for obtaining the maximum clock frequency ([3, 9, 2]).

However, there can be paths which are sensitizable but do not affect the clock period.

In this chapter, we introduce a class of such paths called *multi-clock paths*. Multi-clock paths are combinational paths whose delay time can be greater than the clock period. We discuss the properties of multi-clock paths and propose a method for detecting such paths in given sequential circuits. Multi-clock paths lie in circuits typically due to the operations controlled by *waiting states*, where the propagation of signals are waited on more than one clock cycles, and the delay time of the path can be greater than the clock period. We show a method to detect multi-clock paths by computing the interval of value changes between each pair of registers.

The rest of this chapter is organized as follows. In the next section, we show definitions of finite state machines. In Section 3, we define multi-clock paths. Section 4 contains a method of analyzing update cycles of registers. In Section 5, we show a method to calculate interval of value changes between registers and to find multi-clock paths. Section 6 is a conclusion.

5.2 Finite State Machines

In this section, we show a definition related to finite state machines (FSMs.)

Definition 5.1 An FSM M is a 6-tuple $(S, \Sigma, \Gamma, \delta, \lambda, q_0)$ where

- S is a finite set of states,
- Σ is an input alphabet,
- Γ is an output alphabet,
- $\delta : S \times \Sigma \to S$ is a state transition function,
- $\lambda : S \times \Sigma \to \Gamma$ is an output function,
- q_0 $(\in S)$ is the initial state.

□

The behavior of M with respect to an input sequence $a_1 a_2 \ldots a_n$ $(a_i \in \Sigma)$ is a sequence of states $q_0 q_1 \ldots q_n$ $(q_i \in S)$ and a sequence of outputs $o_1 o_2 \ldots o_n$ $(o_i \in \Gamma)$, where each of the states and the outputs satisfies $q_i = \delta(q_{i-1}, a_i)$ and $o_i = \lambda(q_{i-1}, a_i)$.

Let Σ^* be a set of all input sequences over Σ, and let Σ^k be a set of input sequences with length k. The symbol ε denotes the sequence with length 0.

To represent the behavior of M, the domain of δ and λ are extended and $\delta^* : S \times \Sigma^* \to S$ and $\lambda^* : S \times \Sigma^* \to \Gamma^*$ are introduced as below. The operation '$\cdot$' is the concatenation of strings.

Definition 5.2 $\delta^* : S \times \Sigma^* \to S$ is defined as follows.

- $\delta^*(q, \varepsilon) = q$
- $\delta^*(q, xa) = \delta(\delta^*(q, x), a) \quad (a \in \Sigma, x \in \Sigma^*)$

□

Definition 5.3 $\lambda^* : S \times \Sigma^* \to \Gamma^*$ is defined as follows.

- $\lambda^*(q, \varepsilon) = \varepsilon$
- $\lambda^*(q, xa) = \lambda^*(q, x) \cdot \lambda(\delta^*(q, x), a) \quad (a \in \Sigma, x \in \Sigma^*)$

□

An FSM is implemented as a sequential logic circuit with a set of registers (flip-flops) and a combinational circuit. States of the FSM is encoded into binary vectors and represented as the set of values of the registers. The combinational circuit calculates the encoded state transition function and the output function.

As the output functions, we only deal with the set of encoded state transition functions themselves. We denote the transition function for each register r as an output function λ_r.

5.3 Multi-Clock Paths in Finite State Machines

In this section, we describe the notion of multi-clock paths. Because we use the FSM model, we do not deal with individual paths in combinational logic of sequential machines. We refer to 'the set of all paths from register r_{in} to register r_{out}' simply as a path from r_{in} to r_{out}. In addition, we are concerned with *changes of logic values* and not with *events* on registers. This means that we deal with hazard-free circuits.

Let us consider a path from register r_{in} to r_{out}. The number of clock cycles from value change of r_{in} to that of r_{out} can be different according to the states. The minimum number of the cycles decides the number of clocks the path is allowed to spend for value propagation. We define the number as the interval of value changes of the path.

Definition 5.4 The interval of value changes of the path from register r_{in} to register r_{out} is said to be k, when k is the maximum number that satisfies the next relation:

"for any input sequence and for any state,

when the value of r_{in} has just changed,

the value of r_{out} does not change for k cycles."

The path from r_{in} to r_{out} is called a multi-clock path when this relation holds for some $k > 1$. □

Multi-clock paths are the paths in combinational circuits which are sensitizable but do not affect the decision of the minimum clock period.

When the interval of value changes of a path is k, the path is allowed to spend k clock cycles. That is, the timing constraint of the path is

$$\text{(delay of the path)} \leq k \times \text{(clock period)}.$$

The value k of each path can be computed as follows:

1. Update cycle analysis of registers:

   Checking whether the value of registers has been changed or not at each state, and computing the update cycle.

2. Timing analysis of each path:

   Analyzing the interval of value changes on each path using the update cycles of the input and the output registers.

In the following two sections, we describe formalization and algorithms for these two processes.

5.4 Update Cycle Analysis of Registers

In the following, we fix a register r and introduce a set of states where the value of r does not change during k clocks.

First, let RS be the set of reachable states from the initial state of FSM.

Definition 5.5 The set of reachable states from the initial state RS is defined as follows.

$$RS = \left\{ q \mid \exists x \in \Sigma^*,\ q = \delta^*(q_0,\ x) \right\}$$

□

Let S_0 be the set of initial states of FSM. From our FSM definition, $S_0 = \{q_0\}$. RS is obtained with the procedure shown in Figure 5.1. In this procedure, PS represents the set of present states and NS represents the set of next states of PS. States are traversed until the set of newly visited states (the *frontier*) becomes $\emptyset$.

The procedure is executed using a symbolic state traversal of finite state machines using OBDDs ([14, 10, 7]). In the symbolic state traversal, primary inputs and registers are represented as logical variables. The sets S_0, PS, NS, RS and the function δ are all described as logic functions, i.e., OBDDs representing these functions. The manipulations of state sets such as $\cap$, $\cup$, $-$ are executed as logic operations such as AND, OR, and NOT.

```
procedure ReachableStates(S_0)                                    1
    S_0: the set of the initial states of FSM                     2
    PS:  the set of present states                                3
    NS:  the set of next states of PS                             4
    RS:  the set of reachable states from S_0                     5
begin                                                             6
    PS := S_0;                                                    7
    RS := S_0;                                                    8
    while (PS ≠ ∅) do                                             9
        NS := {δ(q,a) | q ∈ PS, a ∈ Σ};                           10
        PS := NS \ RS;                                            11
        RS := RS ∪ NS;                                            12
    end while ;                                                   13
    return RS;                                                    14
end                                                               15
```

Figure 5.1: Analysis of reachable states from the initial state.

Next, we define state sets RS^r_k ($\subseteq RS$) where the value of register r does not change during k clock cycles. In other words, when starting from a state in RS^r_k, the value of the register does not change for all possible input sequence with length k. Formally, state sets RS^r_k is defined as follows.

Definition 5.6 The set of states where register r does not change during k clock cycles, RS^r_k, is:

$$RS^r_k = \left\{ q \mid q \in RS,\, \forall x \in \Sigma^k,\, \lambda^*_r(q,x) \in \{0^k, 1^k\} \right\}.$$

□

Proposition 5.1 RS^r_k $(k = 0, 1, \ldots)$ have the following property.

$$RS = RS^r_0 = RS^r_1 \supseteq RS^r_2 \supseteq RS^r_3 \supseteq RS^r_4 \ \ldots$$

□

Let $\mathcal{K}_r$ be the maximum number of k such that $RS^r_{k-1} \neq RS^r_k$. Note that, if there is a strongly connected sink component in the transition relation, where the value of r is the same in all the states, then $RS^r_k = RS^r_{k+1} \neq \emptyset$ with some k. Otherwise, $RS^r_k = \emptyset$ with some k.

Lemma 5.1 The following formula holds with $k > 2$.

$$RS^r_k = \{q \mid q \in RS^r_{k-1},\, \forall a \in \Sigma,\, \delta(q,a) \in RS^r_{k-1}\}$$

Proof: ($\supseteq$) Let q be an element of the set of the right side. Since $q \in RS^r_{k-1}$, $\lambda^*_r(q,x) \in \{0^{k-1}, 1^{k-1}\}$ for all $x \in \Sigma^{k-1}$. Since $q' = \delta(q,a) \in RS^r_{k-1}$, $\lambda^*_r(q',x) \in \{0^{k-1}, 1^{k-1}\}$ for all $x \in \Sigma^{k-1}$. Thus, for any $w \in \Sigma^k$, there exist $a \in \Sigma$ and $x \in \Sigma^{k-1}$ s.t. $w = ax$ and $\lambda^*_r(q,w) \in \{0^k, 1^k\}$. These lead to $q \in RS^r_k$.

($\subseteq$) Let q be an element of the set RS^r_k, then $q \in RS^r_{k-1}$. From the definition of RS^r_k, for any $ax \in \Sigma^k$, $\lambda^*_r(\delta(q,a),x) \in \{0^{k-1}, 1^{k-1}\}$. Hence, for any $a \in \Sigma$, $\delta(q,a) \in RS^r_{k-1}$. □

We show a procedure to compute the state sets RS_k^r for all $k = 1, 2, \ldots, \mathcal{K}_r$ based on Lemma 5.1.

Step 1. Compute the set RS of reachable states from the initial state of FSM, and then let

$$RS_1^r := RS.$$

Step 2. Using the state set RS, compute the state set RS_2^r, where

$$RS_2^r := \{q \mid q \in RS,\ \forall a_1 a_2 \in \Sigma^2, \lambda_r(q, a_1) = \lambda_r(\delta(q, a_1), a_2)\}.$$

Step 3. Using RS_2^r, RS_k^r ($k \geq 3$) are computed as follows. We also compute $\mathcal{K}_r$.

```
k := 3;                                                          1
while (RS^r_{k-2} ≠ RS^r_{k-1}) do                               2
    RS^r_k := {q | q ∈ RS^r_{k-1} ∧ ∀a ∈ Σ : δ(q,a) ∈ RS^r_{k-1}}  3
    k := k + 1;                                                  4
end while                                                        5
K_r := k − 2;                                                    6
```

First, state sets RS_1^r and RS_2^r are computed using symbolic state traversal in step 1 and step 2. Secondly, state set RS_3^r is computed from RS_2^r, and similarly, we can obtain state sets RS_4^r, RS_5^r, ... with Lemma 5.1 in step 3. Note that RS_k^r is computed until $RS_{k-2}^r = RS_{k-1}^r$.

On the other hand, we define the set CS_r of states where the value of register r has just changed. Formally, state set CS_r is defined as follows.

Definition 5.7 The set of states where register r has just changed, CS_r, is:

$$CS_r = \{q \mid \exists a_1 a_2 \in \Sigma^2, \exists q' \in RS, \text{ s.t.}$$

$$q = \delta(q', a_1),\ \lambda_r(q', a_1) \neq \lambda_r(q, a_2)\}.$$

□

The CS_r can be computed similarly based on the symbolic state traversal.

5.5 Detection of Multi-Clock Paths

The interval of value changes in Definition 5.4 can be computed as follows.

Proposition 5.2 Let r_{in} and r_{out} be registers. The interval of value changes from r_{in} to r_{out} is k, when k is the maximum number satisfying the following condition.

$$CS_{r_{in}} \subseteq RS_k^{r_{out}}$$

□

The number k is computed as the procedure shown in Figure 5.2, where i is incremented from 2 to $\mathcal{K}_{r_{out}}$ while $CS_{r_{in}} \not\subseteq RS_i^{r_{out}}$.

If k is equal to $\mathcal{K}_{r_{out}}$, then the interval of value changes of the path is infinite. Note that this is the case when the destination register of the path never changes.

If the interval of value changes on a paths is $k \geq 2$, then the path is a multi-clock path, and the allowable delay time of the path is the clock period multiplied by k.

The procedure for finding all multi-clock paths is as follows.

Step 1. Compute the reachable state set RS.

Step 2. For each register r, compute CS_r.

Step 3. For each register r, compute RS_k^r, for $k = 2, 3, \ldots$.

Step 4. For each register pair (r_{in}, r_{out}),
compute `Interval`(r_{in}, r_{out}) as shown in Figure 5.2.

Let us evaluate the computation time of this procedure when it is implemented based on symbolic state traversal using OBDD. Let n be the number of registers, and *#repetition* be the number of repetitions in computing the reachable state set. The computation of

```
procedure Interval(r_in, r_out)                                          1
    r_in, r_out : the registers with paths from r_in to r_out            2
    CS_{r_in} : the state set where the value of r_in changes            3
    RS_i^{r_out} : the state set where the value of r_out does not change
          during i clock cycles                                          4
    k : the interval of value changes from r_in to r_out                 5
begin                                                                    6
    k := 2;                                                              7
    while (CS_{r_in} ⊆ RS_k^{r_out}) do                                  8
        if (k = K_{r_out}) then return ∞;                                9
        k := k + 1;                                                      10
    end while                                                            11
    return k − 1;                                                        12
end                                                                      13
```

Figure 5.2: Analysis of the interval of value changes between registers.

RS in Step 1. is that of the usual symbolic state traversal based on OBDD, and the computation time is proportional to the *#repetition*. The computation time of Step 2. is proportional to n, and each CS_r can be obtained with one OBDD operation on RS. Step 3. is executed within $n \times$ *#repetition* steps, since k of RS_k^r is bounded by the *#repetition*. Step 4. is executed within $n^2 \times$ *#repetition* steps.

5.6 Conclusion

In this chapter, we have shown a method to detect multi-clock paths in sequential circuits. Multi-clock paths exist when input and output registers of the paths are guarded with waiting states. The delay time of the path can be greater than the clock period.

In the detection of multi-clock paths, we use the symbolic state traversal to obtain the state sets with update cycle more than 2.

The method proposed in this chapter can be applied to maximize the clock frequency of sequential logic circuits and to optimize the delay of some operations in logic synthesis systems.

Chapter 6

Conclusion

In this thesis, we discussed design and analysis methods of VLSI logic circuits using graph-based representations of Boolean circuits and functions.

In Chapter 2, We have proposed two algorithms for the minimum cut linear arrangement of p-q dags representing adder tree structures. The algorithms give systematic methods to construct VLSI layout of adder trees of any size using any counter component.

The first algorithm finds out an exact minimum solution through the dynamic programming approach. We proved theorems showing that the minimum cut of a complete p-q dag is attained by an arrangement of almost planar graphs with small 'slippage' between each depths. We can reduce the search space based on these theorems. For fixed p and q, the algorithm calculates a solution within time and space $O(n^{\log_{p/q}(2p+q)})$ where n is the size of a given graph. The second algorithm is an approximation algorithm which calculates an arrangement with $O(\log n)$ cutwidth. This algorithm runs in $O(n \log n)$ time.

There is another application of the linear arrangement problems. In some approaches to VLSI circuit design, the logic elements are linearly arranged with small cutwidth, and then folded into a rectangular region. In the standard-cell method, for example, the logic elements are the standard cells each of which occupies rectangular area and usually has

the same height. In layout systems based on this scheme, at first the cells are placed in horizontal rows, and then the wires are routed in the channels between the rows. Development of these methods motivates the study of linear arrangement problems to abstract VLSI layout problems.

In Chapter 3, we discussed the expressive power of graph-based representations of Boolean functions. We introduced Nondeterministic OBDDs and observed that some applications of OBDDs can be viewed as utilizing the power of nondeterminism in order to reduce the size of OBDDs. We showed that the cutwidth of combinational circuits is related to the size of Nondeterministic OBDDs, so, methods for circuit partitioning can be applied to the reduction of Nondeterministic OBDD size. We also showed that OBDDs representing sum-of-product form can be regarded as a restricted form of Nondeterministic OBDDs.

We have shown the potential of Nondeterministic OBDDs, but the size of Nondeterministic OBDDs is affected heavily by how to introduce the nondeterministic variables. We have to develop methods for the construction to make full use of the effect. Some of the relationships of the language classes defined in the chapter are not settled. These are also interesting research topics in future.

In Chapter 4, we focused on design methods for PTL circuits based on graph-based representations. In order to evaluate the merit to employ decomposed FBDDs instead of decomposed OBDDs, we showed statistics of the size of minimum OBDDs and minimum FBDDs for all functions of fixed number of variables. The results can be used for PTL synthesis as cell libraries.

We also applied the exact minimization algorithm to benchmark circuits. Our possible future work includes improvement of FBDD minimization algorithms, investigation of other classes of decision graphs, and, development of decomposition algorithm suitable for FBDDs.

In Chapter 5, we described a method for timing analysis of sequential circuits with multi-clock operations, as an application of OBDD-based state traversal. Multi-clock

paths are typically due to the operations controlled by waiting states, where the propagation of signals are waited on more than one clock cycles. We showed a method to detect multi-clock paths by computing the interval of value changes between each pair of registers. The delay time of multi-clock paths can be greater than the clock period. The information of multi-clock paths can be used to maximize the clock frequency through redesign of the circuit in higher level, or automatic timing-driven optimization by logic synthesis systems.

Bibliography

[1] S. B. Akers. Binary decision diagrams. *IEEE Trans. Computers*, C-27(6):506–516, Jun 1978.

[2] P. Ashar, S. Dey, and S. Malik. Exploiting multicycle false paths in the performance optimization of sequential logic circuits. *IEEE Trans. Computer-Aided Design*, 14(9):1067–1075, Sept. 1995.

[3] J. Benkowski, E. V. Meersch, L. J. M. Claesen, and H. D. Man. Timing verification using statistically sensitizing path. *IEEE Trans. Computer-Aided Design*, 9(10):1073 1084, Oct. 1990.

[4] C. L. Berman. Circuit width, register allocation, and ordered binary decision diagrams. *IEEE Trans. Computer-Aided Design*, 10(8):1059–1066, Aug. 1991.

[5] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, C-35(8):677–691, Aug. 1986.

[6] R. E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Trans. Computers*, 40(2):205–213, Feb 1991.

[7] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, Sept. 1992.

[8] P. Buch, A. Narayan, A. R. Newton, and A. Sangiovanni-Vincentelli. Logic synthesis for large pass transistor circuits. In *Proc. IEEE/ACM International Conference on Computer Aided Design*, pages 663–670, Nov. 1997.

[9] H. C. Chen and D. H. Du. Path sensitization in critical path problem. In *Proc. IEEE/ACM International Conference on Computer Aided Design*, pages 208–221, 1991.

[10] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines based on symbolic execution. In *Automatic Verification Methods for Finite State Systems, LNCS 407*, pages 365–373, Sept. 1989.

[11] O. Coudert and J. C. Madre. Implicit and incremental computation of primes and essential primes of Boolean functions. In *Proc. 29th ACM/IEEE Design Automation Conference*, pages 36–39, Jun 1992.

[12] P. W. Dymond and S. A. Cook. Complexity theory of parallel time and hardware. *Information and Computation*, 80:205–226, 1989.

[13] S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Trans. Computers*, C-39(5):710–713, May 1990.

[14] M. Fujita and E. M. Clarke. Application of BDD to CAD for digital systems. *Information Processing*, 34(5):609–616, May 1993.

[15] M. R. Garey and D. S. Johnson. *COMPUTERS AND INTRACTABILITY — A Guide to the Theory of NP-Completeness.* W.H.Freenan and Co., 1979.

[16] J. Gergov and C. Meinel. Efficient boolean manipulation with OBDDs can be extended to FBDDs. *IEEE Trans. Computers*, 43(10):1197–1209, Oct. 1994.

[17] E. M. Gurari and I. H. Sudborough. Improved dynamic programming algorithm for bandwidth minimization and the mincut linear arrangement problem. *J. Algorithms*, 5:531 546, 1984.

[18] K. Hamaguchi and S. Yajima. On a relation between the complexity of equivalence checking problem and circuit structure for combinational circuits. In *IEICE National Convention Record, Spring'93 SA-2-3*, Mar 1993. (in Japanese).

[19] N. Ishiura and S. Yajima. A class of logic functions expressible by a polynomial-size binary decision diagram. In *Proc. Synthesis and Simulation Meeting and International Interchange (SASIMI '90)*, pages 48–54, Oct. 1990.

[20] T. Lengauer. Upper and lower bounds on the complexity of the min-cut linear arrangement problem on trees. *SIAM J. Alg. Disc. Meth.*, 3(1):99–113, Mar. 1982.

[21] F. S. Makedon, C. H. Papadimitriou, and I. H. Sudborough. Topological bandwidth. *SIAM J. Alg. Disc. Meth.*, 6(3):418–444, July 1985.

[22] Y. Manabe, K. Hagihira, and N. Tokura. The minimum bisection widths of the cube-connected-cycles graph and cube graph. *Trans. IEICE*, J67-D(6):647–654, 1984.

[23] K. L. McMillan. *Symbolic Model Checking - An approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, 1992.

[24] C. Meinel. *Modified Branching Programs and Their Computational Power.* LNCS 370. Springer-Verlag, 1989.

[25] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. 30th ACM/IEEE Design Automation Conference*, pages 272 277, Jun 1993.

[26] S. Muroga. *Logic Design and Switching Theory.* John Wiley & Sons, Inc., 1979.

[27] K. Nitta. Expressive power of binary decision diagrams representing boolean formulas, 1995. Master Thesis, Department of Information Science, Kyoto University.

[28] K. F. Pang, H.-W. Soong, R. Sexton, and P.-H. Ang. Generation of high speed CMOS multiplier-accumulators. In *Proc. 1988 IEEE ICCD*, pages 217–220, Oct. 1988.

[29] A. Parameswar, H. Hara, and T. Sakurai. A high-speed, low-power and swing restored pass-transistor logic based multiply and accumulate circuit for multimedia applications. *IEEE J. Solid-State Circuits*, 31(6):804–809, June 1996.

[30] N. Pippenger. On simultaneous resource bounds. In *Proc. 20th FOCS*, pages 307–311, 1979.

[31] P. Reusens, W. H. Ku, and Y.-H. Mao. Fixed-point high-speed parallel multipliers. In *VLSI Systems and Computations*, pages 301–310. Computer Science Press, 1981.

[32] H. Sawada, T. Suyama, and A. Nagoya. Logic synthesis for look-up table based FPGAs using functional decomposition and boolean resubstitution. *IEICE Trans. Inf. & Syst.*, E80-D(10):1017–1023, Oct. 1997.

[33] H. Sawada, Y. Takenaga, and S. Yajima. On the computational power of Binary Decision Diagrams. *IEICE Trans. Inf. & Syst.*, E77-D(6):611–618, Jun 1994.

[34] P. J. Song and G. De Micheli. Circuit and architecture trade-offs for high-speed multiplication. *IEEE J. Solid-State Circuits*, 26(9):1184–1198, Sept. 1991.

[35] W. J. Stenzel, W. J. Kubitz, and G. H. Garcia. A compact high-speed parallel multiplication scheme. *IEEE Trans. Computers*, C-26(10):948–957, Oct. 1977.

[36] M. Suzuki, N. Ohkubo, T. Shinbo, T. Yamanaka, A. Shimizu, K. Sasaki, and Y. Nakagome. A 1.5-ns 32-b CMOS ALU in double pass-transistor logic. *IEEE J. Solid-State Circuits*, 28(11):1145–1151, Nov. 1993.

[37] M. Tachibana. Heuristic algorithms for FBDD node minimization with application to pass-transistor-logic and dcvs synthesis. In *Proc. Synthesis and Simulation Meeting and International Interchange (SASIMI '96)*, pages 96–101, Nov. 1996.

[38] K. Taki and B. Lee. Low power pass-transistor logic and application examples. *IEICE Trans.*, J80-A(5):753–764, May 1997. (invited), (in Japanese).

[39] J. D. Ullman. *Computational Aspects of VLSI*. Computer Science Press, 1984.

[40] J. Vuillemin. A very fast multiplication algorithm for VLSI implementation. *Integration*, 1:39–52, 1983.

[41] C. S. Wallace. A suggestion for a fast multiplier. *IEEE Trans. Computers*, pages 14–17, Feb. 1964.

[42] M. Yannakakis. A polynomial algorithm for the min-cut linear arrangement of trees. *J. ACM*, 32(4):950–988, Oct. 1985.

[43] K. Yano, Y. Sasaki, K. Rikino, and K. Seki. Top-down pass-transistor logic design. *IEEE J. Solid-State Circuits*, 31(6):792–803, June 1996.

[44] K. Yano, T. Yamanaka, T. Nishida, M. Saito, K. Shimohigashi, and A. Shimizu. A 3.8-ns CMOS 16 × 16-b multiplier using complementary pass-transistor logic. *IEEE J. Solid-State Circuits*, 25(2):388–395, Apr. 1990.

[45] K. Yasuoka. A new method to represent sets of products: Ternary decision diagrams. *IEICE Trans. Fundamentals*, E78-A(12), Dec 1995.

[46] D. Zuras and W. H. McAllister. Balanced delay trees and combinatorial division in VLSI. *IEEE J. Solid-State Circuits*, 21(5):814–819, Oct. 1986.

Acknowledgment

I would like to appreciate Professor Kazuo Iwama of Kyoto University for his guidance in preparing the thesis and for his helpful comments. Thanks are also due to Professor Yahiko Kambayashi and Professor Shinji Tomita of Kyoto University for their interesting comments on the draft of the thesis.

I would also like to express my sincere appreciation to Professor Shuzo Yajima of Kansai University for his continuous guidance, interesting suggestions, accurate criticism and encouragements throughout this research.

I would also like to thank Professor Katsumasa Watanabe of Nara Institute of Science and Technology who has been giving me interesting comments and encouragements.

I would also like to express my gratitude to Professor Naofumi Takagi of Nagoya University. He introduced me to this research field and has been giving me invaluable suggestions and encouragements.

I am indebted to Associate Professor Shinji Kimura of Nara Institute of Science and Technology not only for discussions and collaboration during this research, but also for reading drafts of this thesis thoroughly and suggesting improvements.

I would also like to express my thanks to Associate Professor Kiyoharu Hamaguchi of Osaka University and Associate Professor Yasuhiko Takenaga of the University of Electro-Communications for their valuable suggestions and discussions.

I would also like to thank Mr. Koyo Nitta of NTT LSI Laboratories and Mr. Hironori Bouno of Hewlett Packard Laboratories Japan for their great contribution to the proofs

in Section 3.5. I also wish to thank Mr. Hiroshi Hatakeda and Mr. Kazuhiro Nakamura of Nara Institute of Science and Technology for their valuable discussions on the studies of Chapter 4 and Chapter 5 respectively.

I also acknowledge the interesting comments that I have received from Associate Professor Hiroyuki Ochi of Hiroshima City University, Associate Professor Koichi Yasuoka and Mr. Takashi Horiyama of Kyoto University.

I am grateful to Mr. Hiroshi Sawada of NTT Communications Science Laboratories who provided programs for experiments utilized in Section 4.5.

Thanks are also due to all the members of Professor Watanabe's Laboratory for their discussions and supports through this research.

List of Publications by the Author

Major Publications

1. K. Takagi, K. Nitta, H. Bouno, Y. Takenaga and S. Yajima. Computational Power of Nondeterministic Ordered Binary Decision Diagrams and Their Subclasses. *IEICE Trans. Fundamentals*, Vol. E80-A, No. 4, 1997.

2. K. Nakamura, S. Kimura, K. Takagi and K. Watanabe. Timing Verification of Sequential Logic Circuits Based on Controlled Multi-clock Path Analysis. *IEICE Trans. Fundamentals*, Vol. E81-A, No. 12, 1998.

3. K. Takagi and N. Takagi. Minimum Cut Linear Arrangement of *p-q* Dags for VLSI Layout of Adder Trees. To appear in *IEICE Trans. Fundamentals*, Vol. E82-A, No. 5, 1999.

4. K. Takagi, H. Hatakeda, S. Kimura and K. Watanabe. Exact Minimization of FBDDs for Pass-Transistor Logic Optimization. *Proceedings of the 8th Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI '98)*, 1998.

5. K. Nakamura, K. Takagi, S. Kimura and K. Watanabe. Waiting False Path Analysis of Sequential Logic Circuits for Performance Optimization. *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD '98)*, 1998.

Technical Reports

1. K. Takagi, Y. Takenaga and S. Yajima. Memory-Based Parallel Algorithms for Reachability and Satisfiability Problems. In *Report on Technical Group on Algorithms*, IPSJ, 91-AL-21-3, 1991 (in Japanese.)

2. K. Takagi, Y. Takenaga and S. Yajima. Memory-Based Parallel Algorithms for Reachability and Satisfiability Problems. In *Jouhou Kiso Riron Workshop*, 1991 (in Japanese.)

3. H. Morita, K. Ohta and K. Takagi. Is DES Closed? – Another Evidence by Switching Closure Test –. In *Technical Report of IEICE*, ISEC92-49, 1992 (in Japanese.)

4. K. Takagi, N. Takagi and S. Yajima. On Linear Arrangement Algorithms for Directed Acyclic Graphs. In *Technical Report of IEICE*, COMP93-18, SS93-12, 1993.

5. K. Takagi, N. Takagi and S. Yajima. Algorithms for Linear Arrangement Problems of Directed Acyclic Graphs. In *Jouhou Kiso Riron Workshop*, 1993.

6. K. Takagi, N. Takagi and S. Yajima. Linear Arrangement Algorithms for VLSI Layout of High-Speed Multipliers. In *KUIS Technical Report*, KUIS-93-0011, Kyoto University, 1993.

7. K. Takagi and S. Yajima. On the Expressive Power of Polynomial Size OBDDs with Parametric Variables. In *Jouhou Kiso Riron Workshop*, 1994.

8. K. Takagi and S. Yajima. On the Expressive Power of OBDDs with Parametric Variables and Bounded Cutwidth Circuits. In *KUIS Technical Report*, KUIS-95-0005, Kyoto University, 1995.

9. S. Kimura, M. Hirao, K. Takagi and K. Watanabe. Timing Analysis of Logic Circuits with Multiple Clock Operations. In *Technical Report of IEICE*, VLD96-33, 1996 (in Japanese.)

10. Y. Itoh, M. Hirao, K. Takagi, S. Kimura and K. Watanabe. Hardware/Software Codesign and Co-operation on General Purpose Coprocessor Using DMA. In *Technical Report of IEICE*, VLD96-98, ICD96-208, 1997 (in Japanese.)

11. K. Nakamura, S. Kimura, K. Takagi and K. Watanabe. Waiting False Path Analysis of Sequential Logic Circuits. In *Technical Report of IEICE*, VLD97-132, ICD97-237, 1998 (in Japanese.)

12. H. Hatakeda, K. Takagi, S. Kimura and K. Watanabe. Exact Minimization of FBDDs for Pass-Transistor Logic Synthesis. In *Technical Report of IEICE*, VLD98-62, ICD98-165, FTS98-89, 1998 (in Japanese.)

Convention Records

1. K. Takagi, Y. Takenaga and S. Yajima. Memory-Based Parallel Algorithms for Reachability and Satisfiability Problems. In *43rd National Convention Record of IPSJ*, 1991 (in Japanese.)

2. K. Takagi, N. Takagi and S. Yajima. A Minimum Cut Linear Arrangement Algorithm for Directed Acyclic Graphs. In *1993 IEICE National Convention Record of Information and System*, D-6, 1993.

3. K. Takagi and S. Yajima. Boolean Functions Represented by Polynomial Size OBDDs with Quantified Variables. In *1994 IEICE National Convention Record of Information and System*, D-4, 1994.

4. Y. Itoh, M. Hirao, K. Takagi, S. Kimura and K. Watanabe. A Hardware/Software Codesign Method for Computer Systems with General Purpose Coprocessor In *1996 Society conference of IEICE*, A-66, 1996 (in Japanese.)

5. M. Hirao, Y. Itoh, K. Takagi, S. Kimura and K. Watanabe. A Scheduling Method of Operations for General Purpose Coprocessor In *1996 Society conference of IEICE*,

A-67, 1996 (in Japanese.)

6. K. Nakamura, S. Kimura, K. Takagi and K. Watanabe. Timing Verification Using Waiting False Path Analysis of Sequential Logic Circuits. In *1998 General conference of IEICE*, A-3-8, 1998 (in Japanese.)

7. H. Hatakeda, K. Takagi, S. Kimura and K. Watanabe. Exact Minimization of FBDDs for Pass-Transistor Logic Synthesis. In *1998 Society conference of IEICE*, A-3-7, 1998 (in Japanese.)